

## AAA—ASCII Adjust After Addition

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
37	AAA	ZO	Invalid	Valid	ASCII adjust AL after addition.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

### Description

Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register increments by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

```

IF 64-Bit Mode
  THEN
    #UD;
  ELSE
    IF ((AL AND 0FH) > 9) or (AF = 1)
      THEN
        AX ← AX + 106H;
        AF ← 1;
        CF ← 1;
      ELSE
        AF ← 0;
        CF ← 0;
    FI;
    AL ← AL AND 0FH;
  FI;

```

### Flags Affected

The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

**Compatibility Mode Exceptions**

Same exceptions as protected mode.

**64-Bit Mode Exceptions**

#UD                      If in 64-bit mode.

## AAD—ASCII Adjust AX Before Division

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
D5 0A	AAD	Z0	Invalid	Valid	ASCII adjust AX before division.
D5 <i>ib</i>	AAD <i>imm8</i>	Z0	Invalid	Valid	Adjust AX before division to number base <i>imm8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AX register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to  $(AL + (10 * AH))$ , and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit (base 10) number in registers AH and AL.

The generalized version of this instruction allows adjustment of two unpacked digits of any number base (see the “Operation” section below), by setting the *imm8* byte to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAD mnemonic is interpreted by all assemblers to mean adjust ASCII (base 10) values. To adjust values in another number base, the instruction must be hand coded in machine code (D5 *imm8*).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

IF 64-Bit Mode

THEN

#UD;

ELSE

tempAL  $\leftarrow$  AL;

tempAH  $\leftarrow$  AH;

AL  $\leftarrow$  (tempAL + (tempAH \* *imm8*) AND FFH;

(\* *imm8* is set to 0AH for the AAD mnemonic.\*)

AH  $\leftarrow$  0;

FI;

The immediate value (*imm8*) is taken from the second byte of the instruction.

### Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register; the OF, AF, and CF flags are undefined.

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as protected mode.

**Compatibility Mode Exceptions**

Same exceptions as protected mode.

**64-Bit Mode Exceptions**

#UD                      If in 64-bit mode.

## AAM—ASCII Adjust AX After Multiply

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
D4 0A	AAM	Z0	Invalid	Valid	ASCII adjust AX after multiply.
D4 <i>ib</i>	AAM <i>imm8</i>	Z0	Invalid	Valid	Adjust AX after multiply to number base <i>imm8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the “Operation” section below). Here, the *imm8* byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 *imm8*).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL ← AL;
        AH ← tempAL / imm8; (* imm8 is set to 0AH for the AAM mnemonic *)
        AL ← tempAL MOD imm8;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

### Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register. The OF, AF, and CF flags are undefined.

### Protected Mode Exceptions

#DE If an immediate value of 0 is used.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

**Compatibility Mode Exceptions**

Same exceptions as protected mode.

**64-Bit Mode Exceptions**

#UD                      If in 64-bit mode.

## AAS—ASCII Adjust AL After Subtraction

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
3F	AAS	Z0	Invalid	Valid	ASCII adjust AL after subtraction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register decrements by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top four bits set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

```

IF 64-bit mode
  THEN
    #UD;
  ELSE
    IF ((AL AND 0FH) > 9) or (AF = 1)
      THEN
        AX ← AX - 6;
        AH ← AH - 1;
        AF ← 1;
        CF ← 1;
        AL ← AL AND 0FH;
      ELSE
        CF ← 0;
        AF ← 0;
        AL ← AL AND 0FH;
    FI;
  FI;

```

### Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as protected mode.

**Compatibility Mode Exceptions**

Same exceptions as protected mode.

**64-Bit Mode Exceptions**

#UD                      If in 64-bit mode.



## ADC—Add with Carry

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
14 <i>ib</i>	ADC AL, <i>imm8</i>	I	Valid	Valid	Add with carry <i>imm8</i> to AL.
15 <i>iw</i>	ADC AX, <i>imm16</i>	I	Valid	Valid	Add with carry <i>imm16</i> to AX.
15 <i>id</i>	ADC EAX, <i>imm32</i>	I	Valid	Valid	Add with carry <i>imm32</i> to EAX.
REX.W + 15 <i>id</i>	ADC RAX, <i>imm32</i>	I	Valid	N.E.	Add with carry <i>imm32</i> sign extended to 64-bits to RAX.
80 /2 <i>ib</i>	ADC <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Add with carry <i>imm8</i> to <i>r/m8</i> .
REX + 80 /2 <i>ib</i>	ADC <i>r/m8</i> , <i>imm8</i>	MI	Valid	N.E.	Add with carry <i>imm8</i> to <i>r/m8</i> .
81 /2 <i>iw</i>	ADC <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Add with carry <i>imm16</i> to <i>r/m16</i> .
81 /2 <i>id</i>	ADC <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Add with CF <i>imm32</i> to <i>r/m32</i> .
REX.W + 81 /2 <i>id</i>	ADC <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Add with CF <i>imm32</i> sign extended to 64-bits to <i>r/m64</i> .
83 /2 <i>ib</i>	ADC <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Add with CF sign-extended <i>imm8</i> to <i>r/m16</i> .
83 /2 <i>ib</i>	ADC <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Add with CF sign-extended <i>imm8</i> into <i>r/m32</i> .
REX.W + 83 /2 <i>ib</i>	ADC <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Add with CF sign-extended <i>imm8</i> into <i>r/m64</i> .
10 <i>/r</i>	ADC <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Add with carry byte register to <i>r/m8</i> .
REX + 10 <i>/r</i>	ADC <i>r/m8</i> , <i>r8</i>	MR	Valid	N.E.	Add with carry byte register to <i>r/m64</i> .
11 <i>/r</i>	ADC <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Add with carry <i>r16</i> to <i>r/m16</i> .
11 <i>/r</i>	ADC <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Add with CF <i>r32</i> to <i>r/m32</i> .
REX.W + 11 <i>/r</i>	ADC <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Add with CF <i>r64</i> to <i>r/m64</i> .
12 <i>/r</i>	ADC <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Add with carry <i>r/m8</i> to byte register.
REX + 12 <i>/r</i>	ADC <i>r8</i> , <i>r/m8</i>	RM	Valid	N.E.	Add with carry <i>r/m64</i> to byte register.
13 <i>/r</i>	ADC <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Add with carry <i>r/m16</i> to <i>r16</i> .
13 <i>/r</i>	ADC <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Add with CF <i>r/m32</i> to <i>r32</i> .
REX.W + 13 <i>/r</i>	ADC <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Add with CF <i>r/m64</i> to <i>r64</i> .

### NOTES:

\*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
MR	ModRM:r/m ( <i>r</i> , <i>w</i> )	ModRM:reg ( <i>r</i> )	NA	NA
MI	ModRM:r/m ( <i>r</i> , <i>w</i> )	<i>imm8</i>	NA	NA
I	AL/AX/EAX/RAX	<i>imm8</i>	NA	NA

### Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

$DEST \leftarrow DEST + SRC + CF;$

## Intel C/C++ Compiler Intrinsic Equivalent

ADC: extern unsigned char \_addcarry\_u8(unsigned char c\_in, unsigned char src1, unsigned char src2, unsigned char \*sum\_out);

ADC: extern unsigned char \_addcarry\_u16(unsigned char c\_in, unsigned short src1, unsigned short src2, unsigned short \*sum\_out);

ADC: extern unsigned char \_addcarry\_u32(unsigned char c\_in, unsigned int src1, unsigned char int, unsigned int \*sum\_out);

ADC: extern unsigned char \_addcarry\_u64(unsigned char c\_in, unsigned \_\_int64 src1, unsigned \_\_int64 src2, unsigned \_\_int64 \*sum\_out);

## Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## ADCX — Unsigned Integer Addition of Two Operands with Carry Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F6 /r ADCX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.
66 REX.w 0F 38 F6 /r ADCX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

### Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the carry-flag (CF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of CF can represent a carry from a previous addition. The instruction sets the CF flag with the carry generated by the unsigned addition of the operands.

The ADCX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we need to make sure the CF is in a desired initial state. Often, this initial state needs to be 0, which can be achieved with an instruction to zero the CF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64 bits.

ADCX executes normally either inside or outside a transaction region.

Note: ADCX defines the OF flag differently than the ADD/ADC instructions as defined in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

### Operation

IF OperandSize is 64-bit

THEN CF:DEST[63:0] ← DEST[63:0] + SRC[63:0] + CF;

ELSE CF:DEST[31:0] ← DEST[31:0] + SRC[31:0] + CF;

FI;

### Flags Affected

CF is updated based on result. OF, SF, ZF, AF and PF flags are unmodified.

### Intel C/C++ Compiler Intrinsic Equivalent

unsigned char \_addcarryx\_u32 (unsigned char c\_in, unsigned int src1, unsigned int src2, unsigned int \*sum\_out);

unsigned char \_addcarryx\_u64 (unsigned char c\_in, unsigned \_\_int64 src1, unsigned \_\_int64 src2, unsigned \_\_int64 \*sum\_out);

### SIMD Floating-Point Exceptions

None

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.

#SS(0) For an illegal address in the SS segment.

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

**Virtual-8086 Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**ADD—Add**

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
04 <i>ib</i>	ADD AL, <i>imm8</i>	I	Valid	Valid	Add <i>imm8</i> to AL.
05 <i>iw</i>	ADD AX, <i>imm16</i>	I	Valid	Valid	Add <i>imm16</i> to AX.
05 <i>id</i>	ADD EAX, <i>imm32</i>	I	Valid	Valid	Add <i>imm32</i> to EAX.
REX.W + 05 <i>id</i>	ADD RAX, <i>imm32</i>	I	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to RAX.
80 /0 <i>ib</i>	ADD r/m8, <i>imm8</i>	MI	Valid	Valid	Add <i>imm8</i> to r/m8.
REX + 80 /0 <i>ib</i>	ADD r/m8*, <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to r/m8.
81 /0 <i>iw</i>	ADD r/m16, <i>imm16</i>	MI	Valid	Valid	Add <i>imm16</i> to r/m16.
81 /0 <i>id</i>	ADD r/m32, <i>imm32</i>	MI	Valid	Valid	Add <i>imm32</i> to r/m32.
REX.W + 81 /0 <i>id</i>	ADD r/m64, <i>imm32</i>	MI	Valid	N.E.	Add <i>imm32</i> sign-extended to 64-bits to r/m64.
83 /0 <i>ib</i>	ADD r/m16, <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to r/m16.
83 /0 <i>ib</i>	ADD r/m32, <i>imm8</i>	MI	Valid	Valid	Add sign-extended <i>imm8</i> to r/m32.
REX.W + 83 /0 <i>ib</i>	ADD r/m64, <i>imm8</i>	MI	Valid	N.E.	Add sign-extended <i>imm8</i> to r/m64.
00 /r	ADD r/m8, r8	MR	Valid	Valid	Add r8 to r/m8.
REX + 00 /r	ADD r/m8*, r8*	MR	Valid	N.E.	Add r8 to r/m8.
01 /r	ADD r/m16, r16	MR	Valid	Valid	Add r16 to r/m16.
01 /r	ADD r/m32, r32	MR	Valid	Valid	Add r32 to r/m32.
REX.W + 01 /r	ADD r/m64, r64	MR	Valid	N.E.	Add r64 to r/m64.
02 /r	ADD r8, r/m8	RM	Valid	Valid	Add r/m8 to r8.
REX + 02 /r	ADD r8*, r/m8*	RM	Valid	N.E.	Add r/m8 to r8.
03 /r	ADD r16, r/m16	RM	Valid	Valid	Add r/m16 to r16.
03 /r	ADD r32, r/m32	RM	Valid	Valid	Add r/m32 to r32.
REX.W + 03 /r	ADD r64, r/m64	RM	Valid	N.E.	Add r/m64 to r64.

**NOTES:**

\*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AX/EAX/RAX	imm8	NA	NA

**Description**

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the CF and OF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

DEST  $\leftarrow$  DEST + SRC;

## Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## ADDPD—Add Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 58 /r ADDPD xmm1, xmm2/m128	A	V/V	SSE2	Add packed double-precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 58 /r VADDPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed double-precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed double-precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.
EVEX.NDS.128.66.0F.W1 58 /r VADDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Add packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.66.0F.W1 58 /r VADDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Add packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.66.0F.W1 58 /r VADDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Add packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Add two, four or eight packed double-precision floating-point values from the first source operand to the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.



**Operation****VADDPD (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VADDPD (EVEX encoded versions) when src2 operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ← SRC1[i+63:i] + SRC2[63:0]

ELSE

DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VADDPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]

DEST[127:64] ← SRC1[127:64] + SRC2[127:64]

DEST[191:128] ← SRC1[191:128] + SRC2[191:128]

DEST[255:192] ← SRC1[255:192] + SRC2[255:192]

DEST[MAXVL-1:256] ← 0

.

**VADDPD (VEX.128 encoded version)**

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$

**ADDPD (128-bit Legacy SSE version)**

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] + \text{SRC}[63:0]$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] + \text{SRC}[127:64]$   
 $\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VADDPD \_\_m512d \_\_mm512\_add\_pd (\_\_m512d a, \_\_m512d b);  
 VADDPD \_\_m512d \_\_mm512\_mask\_add\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VADDPD \_\_m512d \_\_mm512\_maskz\_add\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);  
 VADDPD \_\_m256d \_\_mm256\_mask\_add\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VADDPD \_\_m256d \_\_mm256\_maskz\_add\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VADDPD \_\_m128d \_\_mm\_mask\_add\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VADDPD \_\_m128d \_\_mm\_maskz\_add\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VADDPD \_\_m512d \_\_mm512\_add\_round\_pd (\_\_m512d a, \_\_m512d b, int);  
 VADDPD \_\_m512d \_\_mm512\_mask\_add\_round\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
 VADDPD \_\_m512d \_\_mm512\_maskz\_add\_round\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b, int);  
 ADDPD \_\_m256d \_\_mm256\_add\_pd (\_\_m256d a, \_\_m256d b);  
 ADDPD \_\_m128d \_\_mm\_add\_pd (\_\_m128d a, \_\_m128d b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## ADDPS—Add Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5B /r ADDPS xmm1, xmm2/m128	A	V/V	SSE	Add packed single-precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEEX.NDS.128.0F.WIG 5B /r VADDPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Add packed single-precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.
VEEX.NDS.256.0F.WIG 5B /r VADDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed single-precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.NDS.128.0F.W0 5B /r VADDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Add packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.0F.W0 5B /r VADDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Add packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.0F.W0 5B /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F	Add packed single-precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Add four, eight or sixteen packed single-precision floating-point values from the first source operand with the second source operand, and stores the packed single-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

## Operation

### VADDPS (EVEX encoded versions) when src2 operand is a register

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

### VADDPS (EVEX encoded versions) when src2 operand is a memory source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**VADDPS (VEX.256 encoded version)**

$DEST[31:0] \leftarrow SRC1[31:0] + SRC2[31:0]$   
 $DEST[63:32] \leftarrow SRC1[63:32] + SRC2[63:32]$   
 $DEST[95:64] \leftarrow SRC1[95:64] + SRC2[95:64]$   
 $DEST[127:96] \leftarrow SRC1[127:96] + SRC2[127:96]$   
 $DEST[159:128] \leftarrow SRC1[159:128] + SRC2[159:128]$   
 $DEST[191:160] \leftarrow SRC1[191:160] + SRC2[191:160]$   
 $DEST[223:192] \leftarrow SRC1[223:192] + SRC2[223:192]$   
 $DEST[255:224] \leftarrow SRC1[255:224] + SRC2[255:224]$   
 $DEST[MAXVL-1:256] \leftarrow 0$

**VADDPS (VEX.128 encoded version)**

$DEST[31:0] \leftarrow SRC1[31:0] + SRC2[31:0]$   
 $DEST[63:32] \leftarrow SRC1[63:32] + SRC2[63:32]$   
 $DEST[95:64] \leftarrow SRC1[95:64] + SRC2[95:64]$   
 $DEST[127:96] \leftarrow SRC1[127:96] + SRC2[127:96]$   
 $DEST[MAXVL-1:128] \leftarrow 0$

**ADDPS (128-bit Legacy SSE version)**

$DEST[31:0] \leftarrow SRC1[31:0] + SRC2[31:0]$   
 $DEST[63:32] \leftarrow SRC1[63:32] + SRC2[63:32]$   
 $DEST[95:64] \leftarrow SRC1[95:64] + SRC2[95:64]$   
 $DEST[127:96] \leftarrow SRC1[127:96] + SRC2[127:96]$   
 $DEST[MAXVL-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

`VADDPS __m512 __mm512_add_ps (__m512 a, __m512 b);`  
`VADDPS __m512 __mm512_mask_add_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);`  
`VADDPS __m512 __mm512_maskz_add_ps (__mmask16 k, __m512 a, __m512 b);`  
`VADDPS __m256 __mm256_mask_add_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);`  
`VADDPS __m256 __mm256_maskz_add_ps (__mmask8 k, __m256 a, __m256 b);`  
`VADDPS __m128 __mm_mask_add_ps (__m128d s, __mmask8 k, __m128 a, __m128 b);`  
`VADDPS __m128 __mm_maskz_add_ps (__mmask8 k, __m128 a, __m128 b);`  
`VADDPS __m512 __mm512_add_round_ps (__m512 a, __m512 b, int);`  
`VADDPS __m512 __mm512_mask_add_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);`  
`VADDPS __m512 __mm512_maskz_add_round_ps (__mmask16 k, __m512 a, __m512 b, int);`  
`ADDPS __m256 __mm256_add_ps (__m256 a, __m256 b);`  
`ADDPS __m128 __mm_add_ps (__m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## ADDSD—Add Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD xmm1, xmm2/m64	A	V/V	SSE2	Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.LIG.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 58 /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Add the low double-precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Adds the low double-precision floating-point values from the second source operand and the first source operand and stores the double-precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VADDSD is encoded with VEX.L=0. Encoding VADDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

### VADDSD (EVEX encoded version)

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

### VADDSD (VEX.128 encoded version)

```

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

### ADDSD (128-bit Legacy SSE version)

```

DEST[63:0] ← DEST[63:0] + SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

VADDSD __m128d _mm_mask_add_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_maskz_add_sd (__mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_add_round_sd (__m128d a, __m128d b, int);
VADDSD __m128d _mm_mask_add_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VADDSD __m128d _mm_maskz_add_round_sd (__mmask8 k, __m128d a, __m128d b, int);
ADDSD __m128d _mm_add_sd (__m128d a, __m128d b);

```

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

## Other Exceptions

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## ADDSS—Add Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS xmm1, xmm2/m32	A	V/V	SSE	Add the low single-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.LIG.F3.0F.WIG 58 /r VADDSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Add the low single-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Add the low single-precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Adds the low single-precision floating-point values from the second source operand and the first source operand, and stores the double-precision floating-point result in the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source and destination operands are the same. Bits (MAXVL-1:32) of the corresponding the destination register remain unchanged.

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VADDSS is encoded with VEX.L=0. Encoding VADDSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.



## Operation

### VADDSS (EVEX encoded versions)

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

### VADDSS DEST, SRC1, SRC2 (VEX.128 encoded version)

```

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

### ADDSS DEST, SRC (128-bit Legacy SSE version)

```

DEST[31:0] ← DEST[31:0] + SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

VADDSS __m128 _mm_mask_add_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_maskz_add_ss (__mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_add_round_ss (__m128 a, __m128 b, int);
VADDSS __m128 _mm_mask_add_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VADDSS __m128 _mm_maskz_add_round_ss (__mmask8 k, __m128 a, __m128 b, int);
ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);

```

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

## Other Exceptions

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

**ADDSUBPD—Packed Double-FP Add/Subtract**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F D0 /r ADDSUBPD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE3	Add/subtract double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG D0 /r VADDSUBPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add/subtract packed double-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG D0 /r VADDSUBPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Add / subtract packed double-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

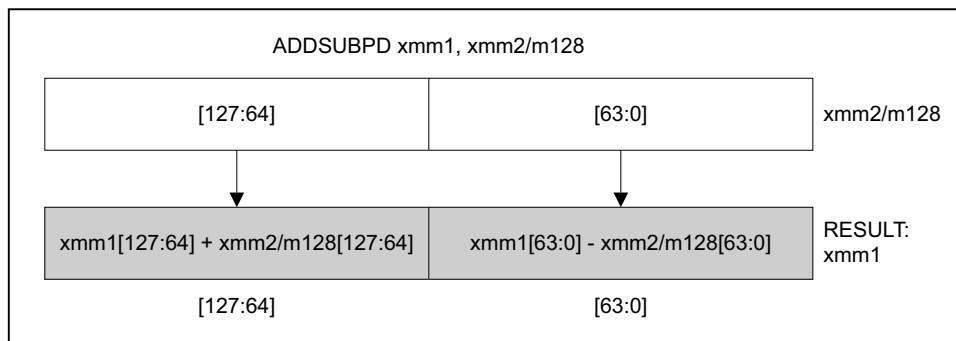
Adds odd-numbered double-precision floating-point values of the first source operand (second operand) with the corresponding double-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered double-precision floating-point values from the second source operand from the corresponding double-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-3.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.



**Figure 3-3. ADDSUBPD—Packed Double-FP Add/Subtract**

### Operation

#### ADDSUBPD (128-bit Legacy SSE version)

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] + \text{SRC}[127:64]$   
 $\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

#### VADDSUBPD (VEX.128 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$

#### VADDSUBPD (VEX.256 encoded version)

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$   
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64]$   
 $\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] - \text{SRC2}[191:128]$   
 $\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] + \text{SRC2}[255:192]$

### Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPD: `__m128d _mm_addsub_pd(__m128d a, __m128d b)`

VADDSUBPD: `__m256d _mm256_addsub_pd (__m256d a, __m256d b)`

### Exceptions

When the source operand is a memory operand, it must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Other Exceptions

See Exceptions Type 2.

## ADDSUBPS—Packed Single-FP Add/Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F D0 /r ADDSUBPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE3	Add/subtract single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG D0 /r VADDSUBPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add/subtract single-precision floating-point values from <i>xmm3/mem</i> to <i>xmm2</i> and stores result in <i>xmm1</i> .
VEX.NDS.256.F2.0F.WIG D0 /r VADDSUBPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX	Add / subtract single-precision floating-point values from <i>ymm3/mem</i> to <i>ymm2</i> and stores result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

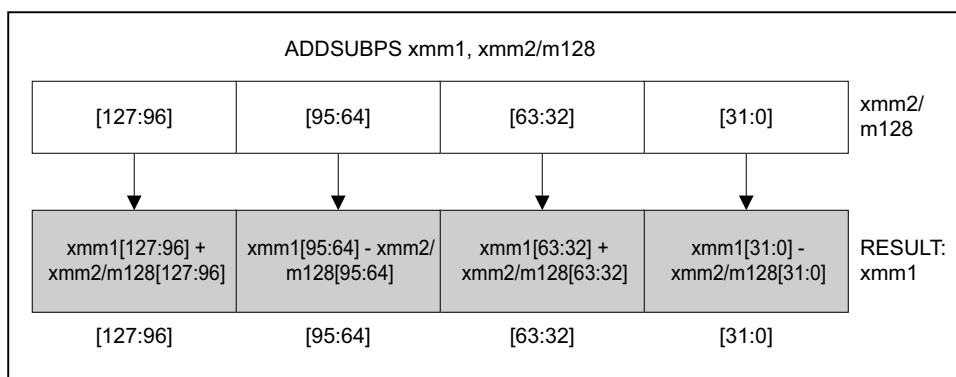
Adds odd-numbered single-precision floating-point values of the first source operand (second operand) with the corresponding single-precision floating-point values from the second source operand (third operand); stores the result in the odd-numbered values of the destination operand (first operand). Subtracts the even-numbered single-precision floating-point values from the second source operand from the corresponding single-precision floating values in the first source operand; stores the result into the even-numbered values of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-4.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.



OM15992

Figure 3-4. ADDSUBPS—Packed Single-FP Add/Subtract

### Operation

#### ADDSUBPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] - \text{SRC}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{DEST}[63:32] + \text{SRC}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{DEST}[95:64] - \text{SRC}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{DEST}[127:96] + \text{SRC}[127:96]$   
 $\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

#### VADDSUBPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$

#### VADDSUBPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] + \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC2}[159:128]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[191:160] + \text{SRC2}[191:160]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC1}[223:192] - \text{SRC2}[223:192]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] + \text{SRC2}[255:224]$

### Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPS: `__m128 _mm_addsub_ps(__m128 a, __m128 b)`

VADDSUBPS: `__m256 _mm256_addsub_ps (__m256 a, __m256 b)`

### Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Exceptions Type 2.

## ADOX — Unsigned Integer Addition of Two Operands with Overflow Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F6 /r ADOX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with OF, r/m32 to r32, writes OF.
F3 REX.w 0F 38 F6 /r ADOX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with OF, r/m64 to r64, writes OF.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

### Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the overflow-flag (OF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of OF represents a carry from a previous addition. The instruction sets the OF flag with the carry generated by the unsigned addition of the operands.

The ADOX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we execute an instruction to zero the OF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64-bits.

ADOX executes normally either inside or outside a transaction region.

Note: ADOX defines the CF and OF flags differently than the ADD/ADC instructions as defined in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

### Operation

IF OperandSize is 64-bit

```
THEN OF:DEST[63:0] ← DEST[63:0] + SRC[63:0] + OF;
ELSE OF:DEST[31:0] ← DEST[31:0] + SRC[31:0] + OF;
```

FI;

### Flags Affected

OF is updated based on result. CF, SF, ZF, AF and PF flags are unmodified.

### Intel C/C++ Compiler Intrinsic Equivalent

```
unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);
```

```
unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);
```

### SIMD Floating-Point Exceptions

None

**Protected Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

**Virtual-8086 Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



**AESDEC—Perform One Round of an AES Decryption Flow**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DE /r AESDEC xmm1, xmm2/m128	RM	V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

This instruction performs a single round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESDEC instruction for all but the last decryption round. For the last decryption round, use the AESDECLAST instruction.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**Operation****AESDEC**

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

**VAESDEC**

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
STATE ← InvMixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)AESDEC: `__m128i _mm_aesdec (__m128i, __m128i)`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

## AESDECLAST—Perform Last Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DF /r AESDECLAST xmm1, xmm2/m128	RM	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from xmm3/m128; store the result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction performs the last round of the AES decryption flow using the Equivalent Inverse Cipher, with the round key from the second source operand, operating on a 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

### Operation

#### AESDECLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

#### VAESDECLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows( STATE );
STATE ← InvSubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDECLAST: `__m128i _mm_aesdeclast (__m128i, __m128i)`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

**AESENC—Perform One Round of an AES Encryption Flow**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DC /r AESENC xmm1, xmm2/m128	RM	V/V	AES	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128-bit round key from the xmm3/m128; store the result in xmm1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

This instruction performs a single round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

Use the AESENC instruction for all but the last encryption rounds. For the last encryption round, use the AESENC-CLAST instruction.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**Operation****AESENC**

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

**VAESENC**

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
STATE ← MixColumns( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)AESENC: `__m128i _mm_aesenc (__m128i, __m128i)`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

## AESENCLAST—Perform Last Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DD /r AESENCLAST xmm1, xmm2/m128	RM	V/V	AES	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
VEX.NDS.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	RVM	V/V	Both AES and AVX flags	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm2 with a 128 bit round key from xmm3/m128; store the result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction performs the last round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

### Operation

#### AESENCLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] (Unmodified)
```

#### VAESENCLAST

```
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows( STATE );
STATE ← SubBytes( STATE );
DEST[127:0] ← STATE XOR RoundKey;
DEST[MAXVL-1:128] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENCLAST: `__m128i _mm_aesenclast (__m128i, __m128i)`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.



**AESIMC—Perform the AES InvMixColumn Transformation**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DB /r AESIMC xmm1, xmm2/m128	RM	V/V	AES	Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.
VEX.128.66.0F38.WIG DB /r VAESIMC xmm1, xmm2/m128	RM	V/V	Both AES and AVX flags	Perform the InvMixColumn transformation on a 128-bit round key from xmm2/m128 and store the result in xmm1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Perform the InvMixColumns transformation on the source operand and store the result in the destination operand. The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

Note: the AESIMC instruction should be applied to the expanded AES round keys (except for the first and last round key) in order to prepare them for decryption using the “Equivalent Inverse Cipher” (defined in FIPS 197).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation****AESIMC**

DEST[127:0] ← InvMixColumns( SRC );

DEST[MAXVL-1:128] (Unmodified)

**VAESIMC**

DEST[127:0] ← InvMixColumns( SRC );

DEST[MAXVL-1:128] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)AESIMC: `__m128i _mm_aesimc (__m128i)`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.vvvv ≠ 1111B.

## AESKEYGENASSIST—AES Round Key Generation Assist

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A DF /r ib AESKEYGENASSIST xmm1, xmm2/m128, imm8	RMI	V/V	AES	Assist in AES round key generation using an 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F3A.WIG DF /r ib VAESKEYGENASSIST xmm1, xmm2/m128, imm8	RMI	V/V	Both AES and AVX flags	Assist in AES round key generation using 8 bits Round Constant (RCON) specified in the immediate byte, operating on 128 bits of data specified in xmm2/m128 and stores the result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

### Description

Assist in expanding the AES cipher key, by computing steps towards generating a round key for encryption, using 128-bit data specified in the source operand and an 8-bit round constant specified as an immediate, store the result in the destination operand.

The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

#### AESKEYGENASSIST

$X3[31:0] \leftarrow \text{SRC}[127:96];$

$X2[31:0] \leftarrow \text{SRC}[95:64];$

$X1[31:0] \leftarrow \text{SRC}[63:32];$

$X0[31:0] \leftarrow \text{SRC}[31:0];$

$\text{RCON}[31:0] \leftarrow \text{ZeroExtend}(\text{Imm8}[7:0]);$

$\text{DEST}[31:0] \leftarrow \text{SubWord}(X1);$

$\text{DEST}[63:32] \leftarrow \text{RotWord}(\text{SubWord}(X1)) \text{ XOR } \text{RCON};$

$\text{DEST}[95:64] \leftarrow \text{SubWord}(X3);$

$\text{DEST}[127:96] \leftarrow \text{RotWord}(\text{SubWord}(X3)) \text{ XOR } \text{RCON};$

$\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}$

**VAESKEYGENASSIST**

$X3[31:0] \leftarrow \text{SRC}[127:96];$   
 $X2[31:0] \leftarrow \text{SRC}[95:64];$   
 $X1[31:0] \leftarrow \text{SRC}[63:32];$   
 $X0[31:0] \leftarrow \text{SRC}[31:0];$   
 $\text{RCON}[31:0] \leftarrow \text{ZeroExtend}(\text{Imm8}[7:0]);$   
 $\text{DEST}[31:0] \leftarrow \text{SubWord}(X1);$   
 $\text{DEST}[63:32] \leftarrow \text{RotWord}(\text{SubWord}(X1)) \text{ XOR } \text{RCON};$   
 $\text{DEST}[95:64] \leftarrow \text{SubWord}(X3);$   
 $\text{DEST}[127:96] \leftarrow \text{RotWord}(\text{SubWord}(X3)) \text{ XOR } \text{RCON};$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0;$

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)AESKEYGENASSIST: `__m128i _mm_aeskeygenassist (__m128i, const int)`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.vvvv  $\neq$  1111B.

## AND—Logical AND

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
24 <i>ib</i>	AND AL, <i>imm8</i>	I	Valid	Valid	AL AND <i>imm8</i> .
25 <i>iw</i>	AND AX, <i>imm16</i>	I	Valid	Valid	AX AND <i>imm16</i> .
25 <i>id</i>	AND EAX, <i>imm32</i>	I	Valid	Valid	EAX AND <i>imm32</i> .
REX.W + 25 <i>id</i>	AND RAX, <i>imm32</i>	I	Valid	N.E.	RAX AND <i>imm32</i> sign-extended to 64-bits.
80 /4 <i>ib</i>	AND <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m8</i> AND <i>imm8</i> .
REX + 80 /4 <i>ib</i>	AND <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m8</i> AND <i>imm8</i> .
81 /4 <i>iw</i>	AND <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	<i>r/m16</i> AND <i>imm16</i> .
81 /4 <i>id</i>	AND <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	<i>r/m32</i> AND <i>imm32</i> .
REX.W + 81 /4 <i>id</i>	AND <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	<i>r/m64</i> AND <i>imm32</i> sign extended to 64-bits.
83 /4 <i>ib</i>	AND <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m16</i> AND <i>imm8</i> (sign-extended).
83 /4 <i>ib</i>	AND <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m32</i> AND <i>imm8</i> (sign-extended).
REX.W + 83 /4 <i>ib</i>	AND <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m64</i> AND <i>imm8</i> (sign-extended).
20 /r	AND <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	<i>r/m8</i> AND <i>r8</i> .
REX + 20 /r	AND <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	<i>r/m64</i> AND <i>r8</i> (sign-extended).
21 /r	AND <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	<i>r/m16</i> AND <i>r16</i> .
21 /r	AND <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	<i>r/m32</i> AND <i>r32</i> .
REX.W + 21 /r	AND <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	<i>r/m64</i> AND <i>r32</i> .
22 /r	AND <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	<i>r8</i> AND <i>r/m8</i> .
REX + 22 /r	AND <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	<i>r/m64</i> AND <i>r8</i> (sign-extended).
23 /r	AND <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	<i>r16</i> AND <i>r/m16</i> .
23 /r	AND <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	<i>r32</i> AND <i>r/m32</i> .
REX.W + 23 /r	AND <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	<i>r64</i> AND <i>r/m64</i> .

### NOTES:

\*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
MR	ModRM:r/m ( <i>r</i> , <i>w</i> )	ModRM:reg ( <i>r</i> )	NA	NA
MI	ModRM:r/m ( <i>r</i> , <i>w</i> )	<i>imm8</i>	NA	NA
I	AL/AX/EAX/RAX	<i>imm8</i>	NA	NA

### Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow the it to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST ← DEST AND SRC;

**Flags Affected**

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

**Protected Mode Exceptions**

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## ANDN — Logical AND NOT

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.0F38.W0 F2 /r ANDN r32a, r32b, r/m32	RVM	V/V	BMI1	Bitwise AND of inverted r32b with r/m32, store result in r32a.
VEX.NDS.LZ.0F38.W1 F2 /r ANDN r64a, r64b, r/m64	RVM	V/NE	BMI1	Bitwise AND of inverted r64b with r/m64, store result in r64a.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical AND of inverted second operand (the first source operand) with the third operand (the second source operand). The result is stored in the first operand (destination operand).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

DEST ← (NOT SRC1) bitwiseAND SRC2;

SF ← DEST[OperandSize - 1];

ZF ← (DEST = 0);

### Flags Affected

SF and ZF are updated based on result. OF and CF flags are cleared. AF and PF flags are undefined.

### Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD If VEX.W = 1.

**ANDPD—Bitwise Logical AND of Packed Double Precision Floating-Point Values**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 54 /r ANDPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F 54 /r VANDPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F 54 /r VANDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 54 /r VANDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 54 /r VANDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 54 /r VANDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical AND of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

**Description**

Performs a bitwise logical AND of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

## Operation

### VANDPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

        THEN

          DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[63:0]

        ELSE

          DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[i+63:i]

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+63:i] remains unchanged\*

      ELSE

        ; zeroing-masking

        DEST[i+63:i] = 0

      FI;

  FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

### VANDPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]

DEST[191:128] ← SRC1[191:128] BITWISE AND SRC2[191:128]

DEST[255:192] ← SRC1[255:192] BITWISE AND SRC2[255:192]

DEST[MAXVL-1:256] ← 0

### VANDPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]

DEST[MAXVL-1:128] ← 0

### ANDPD (128-bit Legacy SSE version)

DEST[63:0] ← DEST[63:0] BITWISE AND SRC[63:0]

DEST[127:64] ← DEST[127:64] BITWISE AND SRC[127:64]

DEST[MAXVL-1:128] (Unmodified)

## Intel C/C++ Compiler Intrinsic Equivalent

VANDPD \_\_m512d \_\_mm512\_and\_pd (\_\_m512d a, \_\_m512d b);

VANDPD \_\_m512d \_\_mm512\_mask\_and\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDPD \_\_m512d \_\_mm512\_maskz\_and\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDPD \_\_m256d \_\_mm256\_mask\_and\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDPD \_\_m256d \_\_mm256\_maskz\_and\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDPD \_\_m128d \_\_mm\_mask\_and\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDPD \_\_m128d \_\_mm\_maskz\_and\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDPD \_\_m256d \_\_mm256\_and\_pd (\_\_m256d a, \_\_m256d b);

ANDPD \_\_m128d \_\_mm\_and\_pd (\_\_m128d a, \_\_m128d b);

## SIMD Floating-Point Exceptions

None



### Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## ANDPS—Bitwise Logical AND of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 54 /r ANDPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.OF 54 /r VANDPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.OF 54 /r VANDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.OF.W0 54 /r VANDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.OF.W0 54 /r VANDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.OF.W0 54 /r VANDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical AND of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation****VANDPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← SRC1[i+31:i] BITWISE AND SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0;

**VANDPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]

DEST[159:128] ← SRC1[159:128] BITWISE AND SRC2[159:128]

DEST[191:160] ← SRC1[191:160] BITWISE AND SRC2[191:160]

DEST[223:192] ← SRC1[223:192] BITWISE AND SRC2[223:192]

DEST[255:224] ← SRC1[255:224] BITWISE AND SRC2[255:224].

DEST[MAXVL-1:256] ← 0;

**VANDPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]

DEST[MAXVL-1:128] ← 0;

**ANDPS (128-bit Legacy SSE version)**

DEST[31:0] ← DEST[31:0] BITWISE AND SRC[31:0]

DEST[63:32] ← DEST[63:32] BITWISE AND SRC[63:32]

DEST[95:64] ← DEST[95:64] BITWISE AND SRC[95:64]

DEST[127:96] ← DEST[127:96] BITWISE AND SRC[127:96]

DEST[MAXVL-1:128] (Unmodified)

### Intel C/C++ Compiler Intrinsic Equivalent

```

VANDPS __m512 __mm512_and_ps (__m512 a, __m512 b);
VANDPS __m512 __mm512_mask_and_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDPS __m512 __mm512_maskz_and_ps (__mmask16 k, __m512 a, __m512 b);
VANDPS __m256 __mm256_mask_and_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDPS __m256 __mm256_maskz_and_ps (__mmask8 k, __m256 a, __m256 b);
VANDPS __m128 __mm_mask_and_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDPS __m128 __mm_maskz_and_ps (__mmask8 k, __m128 a, __m128 b);
VANDPS __m256 __mm256_and_ps (__m256 a, __m256 b);
ANDPS __m128 __mm_and_ps (__m128 a, __m128 b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

**ANDNPD—Bitwise Logical AND NOT of Packed Double Precision Floating-Point Values**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F 55 /r VANDNPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F 55/r VANDNPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 55 /r VANDNPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 55 /r VANDNPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 55 /r VANDNPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical AND NOT of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

**Description**

Performs a bitwise logical AND NOT of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

## Operation

### VANDNPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

            THEN

                DEST[i+63:i] ← (NOT(SRC1[i+63:i])) BITWISE AND SRC2[63:0]

            ELSE

                DEST[i+63:i] ← (NOT(SRC1[i+63:i])) BITWISE AND SRC2[i+63:i]

        FI;

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE ; zeroing-masking

            DEST[i+63:i] = 0

        FI;

    FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

### VANDNPD (VEX.256 encoded version)

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]

DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]

DEST[191:128] ← (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]

DEST[255:192] ← (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]

DEST[MAXVL-1:256] ← 0

### VANDNPD (VEX.128 encoded version)

DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]

DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]

DEST[MAXVL-1:128] ← 0

### ANDNPD (128-bit Legacy SSE version)

DEST[63:0] ← (NOT(DEST[63:0])) BITWISE AND SRC[63:0]

DEST[127:64] ← (NOT(DEST[127:64])) BITWISE AND SRC[127:64]

DEST[MAXVL-1:128] (Unmodified)

## Intel C/C++ Compiler Intrinsic Equivalent

VANDNPD \_\_m512d \_\_mm512\_andnot\_pd (\_\_m512d a, \_\_m512d b);

VANDNPD \_\_m512d \_\_mm512\_mask\_andnot\_pd (\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDNPD \_\_m512d \_\_mm512\_maskz\_andnot\_pd (\_\_mmask8 k, \_\_m512d a, \_\_m512d b);

VANDNPD \_\_m256d \_\_mm256\_mask\_andnot\_pd (\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDNPD \_\_m256d \_\_mm256\_maskz\_andnot\_pd (\_\_mmask8 k, \_\_m256d a, \_\_m256d b);

VANDNPD \_\_m128d \_\_mm\_mask\_andnot\_pd (\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDNPD \_\_m128d \_\_mm\_maskz\_andnot\_pd (\_\_mmask8 k, \_\_m128d a, \_\_m128d b);

VANDNPD \_\_m256d \_\_mm256\_andnot\_pd (\_\_m256d a, \_\_m256d b);

ANDNPD \_\_m128d \_\_mm\_andnot\_pd (\_\_m128d a, \_\_m128d b);

## SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## ANDNPS—Bitwise Logical AND NOT of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 55 /r ANDNPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.OF 55 /r VANDNPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.OF 55 /r VANDNPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.OF.W0 55 /r VANDNPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.OF.W0 55 /r VANDNPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.OF.W0 55 /r VANDNPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical AND NOT of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.



**Operation****VANDNPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]

ELSE

DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE

; zeroing-masking

DEST[i+31:i] = 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VANDNPS (VEX.256 encoded version)**

DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]

DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]

DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]

DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]

DEST[159:128] ← (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]

DEST[191:160] ← (NOT(SRC1[191:160])) BITWISE AND SRC2[191:160]

DEST[223:192] ← (NOT(SRC1[223:192])) BITWISE AND SRC2[223:192]

DEST[255:224] ← (NOT(SRC1[255:224])) BITWISE AND SRC2[255:224].

DEST[MAXVL-1:256] ← 0

**VANDNPS (VEX.128 encoded version)**

DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]

DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]

DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]

DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]

DEST[MAXVL-1:128] ← 0

**ANDNPS (128-bit Legacy SSE version)**

DEST[31:0] ← (NOT(DEST[31:0])) BITWISE AND SRC[31:0]

DEST[63:32] ← (NOT(DEST[63:32])) BITWISE AND SRC[63:32]

DEST[95:64] ← (NOT(DEST[95:64])) BITWISE AND SRC[95:64]

DEST[127:96] ← (NOT(DEST[127:96])) BITWISE AND SRC[127:96]

DEST[MAXVL-1:128] (Unmodified)

### Intel C/C++ Compiler Intrinsic Equivalent

```

VANDNPS __m512 _mm512_andnot_ps (__m512 a, __m512 b);
VANDNPS __m512 _mm512_mask_andnot_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDNPS __m512 _mm512_maskz_andnot_ps (__mmask16 k, __m512 a, __m512 b);
VANDNPS __m256 _mm256_mask_andnot_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDNPS __m256 _mm256_maskz_andnot_ps (__mmask8 k, __m256 a, __m256 b);
VANDNPS __m128 _mm_mask_andnot_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDNPS __m128 _mm_maskz_andnot_ps (__mmask8 k, __m128 a, __m128 b);
VANDNPS __m256 _mm256_andnot_ps (__m256 a, __m256 b);
ANDNPS __m128 _mm_andnot_ps (__m128 a, __m128 b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

**ARPL—Adjust RPL Field of Segment Selector**

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
63 /r	ARPL <i>r/m16, r16</i>	Z0	N. E.	Valid	Adjust RPL of <i>r/m16</i> to not less than RPL of <i>r16</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

**Description**

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program's code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then ensures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program (the segment selector for the application program's code segment can be read from the stack following a procedure call).

This instruction executes as described in compatibility mode and legacy mode. It is not encodable in 64-bit mode.

See "Checking Caller Access Privileges" in Chapter 3, "Protected-Mode Memory Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information about the use of this instruction.

**Operation**

IF 64-BIT MODE

THEN

See MOVSDX;

ELSE

IF DEST[RPL] < SRC[RPL]

THEN

ZF ← 1;

DEST[RPL] ← SRC[RPL];

ELSE

ZF ← 0;

FI;

FI;

**Flags Affected**

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, it is set to 0.

**Protected Mode Exceptions**

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD	The ARPL instruction is not recognized in real-address mode. If the LOCK prefix is used.
-----	---

**Virtual-8086 Mode Exceptions**

#UD	The ARPL instruction is not recognized in virtual-8086 mode. If the LOCK prefix is used.
-----	---

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Not applicable.

## BLENDPD — Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 0D /r ib BLENDPD <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Select packed DP-FP values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 0D /r ib VBLENDPD <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVMI	V/V	AVX	Select packed double-precision floating-point Values from <i>xmm2</i> and <i>xmm3/m128</i> from mask in <i>imm8</i> and store the values in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 0D /r ib VBLENDPD <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Select packed double-precision floating-point Values from <i>ymm2</i> and <i>ymm3/m256</i> from mask in <i>imm8</i> and store the values in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[3:0]

### Description

Double-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [3:0] determine whether the corresponding double-precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is “1”, then the double-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### BLENDPD (128-bit Legacy SSE version)

```
IF (IMM8[0] = 0) THEN DEST[63:0] ← DEST[63:0]
    ELSE DEST [63:0] ← SRC[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← DEST[127:64]
    ELSE DEST [127:64] ← SRC[127:64] FI
DEST[MAXVL-1:128] (Unmodified)
```

#### VBLENDPD (VEX.128 encoded version)

```
IF (IMM8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST [63:0] ← SRC2[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← SRC1[127:64]
    ELSE DEST [127:64] ← SRC2[127:64] FI
DEST[MAXVL-1:128] ← 0
```

**VBLENDPD (VEX.256 encoded version)**

```

IF (IMM8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST [63:0] ← SRC2[63:0] FI
IF (IMM8[1] = 0) THEN DEST[127:64] ← SRC1[127:64]
    ELSE DEST [127:64] ← SRC2[127:64] FI
IF (IMM8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST [191:128] ← SRC2[191:128] FI
IF (IMM8[3] = 0) THEN DEST[255:192] ← SRC1[255:192]
    ELSE DEST [255:192] ← SRC2[255:192] FI

```

**Intel C/C++ Compiler Intrinsic Equivalent**

BLENDPD:     \_\_m128d \_mm\_blend\_pd (\_\_m128d v1, \_\_m128d v2, const int mask);

VBLENDPD:    \_\_m256d \_mm256\_blend\_pd (\_\_m256d a, \_\_m256d b, const int mask);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

## BEXTR — Bit Field Extract

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.OF38.W0 F7 /r BEXTR r32a, r/m32, r32b	RMV	V/V	BMI1	Contiguous bitwise extract from r/m32 using r32b as control; store result in r32a.
VEX.NDS.LZ.OF38.W1 F7 /r BEXTR r64a, r/m64, r64b	RMV	V/N.E.	BMI1	Contiguous bitwise extract from r/m64 using r64b as control; store result in r64a

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

### Description

Extracts contiguous bits from the first source operand (the second operand) using an index value and length value specified in the second source operand (the third operand). Bit 7:0 of the second source operand specifies the starting bit position of bit extraction. A START value exceeding the operand size will not extract any bits from the second source operand. Bit 15:8 of the second source operand specifies the maximum number of bits (LENGTH) beginning at the START position to extract. Only bit positions up to (OperandSize - 1) of the first source operand are extracted. The extracted bits are written to the destination register, starting from the least significant bit. All higher order bits in the destination operand (starting at bit position LENGTH) are zeroed. The destination register is cleared if no bits are extracted.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```
START ← SRC2[7:0];
LEN ← SRC2[15:8];
TEMP ← ZERO_EXTEND_TO_512 (SRC1 );
DEST ← ZERO_EXTEND(TEMP[START+LEN-1: START]);
ZF ← (DEST = 0);
```

### Flags Affected

ZF is updated based on the result. AF, SF, and PF are undefined. All other flags are cleared.

### Intel C/C++ Compiler Intrinsic Equivalent

BEXTR:     unsigned \_\_int32 \_bextr\_u32(unsigned \_\_int32 src, unsigned \_\_int32 start, unsigned \_\_int32 len);

BEXTR:     unsigned \_\_int64 \_bextr\_u64(unsigned \_\_int64 src, unsigned \_\_int32 start, unsigned \_\_int32 len);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD                   If VEX.W = 1.

## BLENDPS — Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 0C /r ib BLENDPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 0C /r ib VBLENDPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Select packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/m128</i> from mask in <i>imm8</i> and store the values in <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 0C /r ib VBLENDPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX	Select packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/m256</i> from mask in <i>imm8</i> and store the values in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Packed single-precision floating-point values from the second source operand (third operand) are conditionally merged with values from the first source operand (second operand) and written to the destination operand (first operand). The immediate bits [7:0] determine whether the corresponding single precision floating-point value in the destination is copied from the second source or first source. If a bit in the mask, corresponding to a word, is “1”, then the single-precision floating-point value in the second source operand is copied, else the value in the first source operand is copied.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### BLENDPS (128-bit Legacy SSE version)

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← DEST[31:0]
    ELSE DEST [31:0] ← SRC[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← DEST[63:32]
    ELSE DEST [63:32] ← SRC[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← DEST[95:64]
    ELSE DEST [95:64] ← SRC[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← DEST[127:96]
    ELSE DEST [127:96] ← SRC[127:96] FI
DEST[MAXVL-1:128] (Unmodified)

```



**VBLENDPS (VEX.128 encoded version)**

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
DEST[MAXVL-1:128] ← 0

```

**VBLENDPS (VEX.256 encoded version)**

```

IF (IMM8[0] = 0) THEN DEST[31:0] ← SRC1[31:0]
    ELSE DEST [31:0] ← SRC2[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← SRC1[63:32]
    ELSE DEST [63:32] ← SRC2[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← SRC1[95:64]
    ELSE DEST [95:64] ← SRC2[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← SRC1[127:96]
    ELSE DEST [127:96] ← SRC2[127:96] FI
IF (IMM8[4] = 0) THEN DEST[159:128] ← SRC1[159:128]
    ELSE DEST [159:128] ← SRC2[159:128] FI
IF (IMM8[5] = 0) THEN DEST[191:160] ← SRC1[191:160]
    ELSE DEST [191:160] ← SRC2[191:160] FI
IF (IMM8[6] = 0) THEN DEST[223:192] ← SRC1[223:192]
    ELSE DEST [223:192] ← SRC2[223:192] FI
IF (IMM8[7] = 0) THEN DEST[255:224] ← SRC1[255:224]
    ELSE DEST [255:224] ← SRC2[255:224] FI.

```

**Intel C/C++ Compiler Intrinsic Equivalent**

BLENDPS:    \_\_m128 \_mm\_blend\_ps (\_\_m128 v1, \_\_m128 v2, const int mask);

VBLENDPS:    \_\_m256 \_mm256\_blend\_ps (\_\_m256 a, \_\_m256 b, const int mask);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

## BLENDVPD — Variable Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 15 /r BLENDVPD <i>xmm1</i> , <i>xmm2/m128</i> , < <i>XMM0</i> >	RM0	V/V	SSE4_1	Select packed DP FP values from <i>xmm1</i> and <i>xmm2</i> from mask specified in <i>XMM0</i> and store the values in <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4B /r /is4 VBLENDVPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>xmm4</i>	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from <i>xmm2</i> or <i>xmm3/m128</i> to <i>xmm1</i> , based on mask bits in the mask operand, <i>xmm4</i> .
VEX.NDS.256.66.0F3A.W0 4B /r /is4 VBLENDVPD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>ymm4</i>	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from <i>ymm2</i> or <i>ymm3/m256</i> to <i>ymm1</i> , based on mask bits in the mask operand, <i>ymm4</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	implicit <i>XMM0</i>	NA
RVMR	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	imm8[7:4]

### Description

Conditionally copy each quadword data element of double-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each quadword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding quadword element in the second source operand, if a mask bit is "1"; or
- the corresponding quadword element in the first source operand, if a mask bit is "0"

The register assignment of the implicit mask operand for BLENDVPD is defined to be the architectural register *XMM0*.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register *XMM0*. An attempt to execute BLENDVPD with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed.

VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPD permits the mask to be any XMM or YMM register. In contrast, BLENDVPD treats *XMM0* implicitly as the mask and do not support non-destructive destination operation.

## Operation

### BLENDVPD (128-bit Legacy SSE version)

MASK  $\leftarrow$  XMM0

IF (MASK[63] = 0) THEN DEST[63:0]  $\leftarrow$  DEST[63:0]

ELSE DEST [63:0]  $\leftarrow$  SRC[63:0] FI

IF (MASK[127] = 0) THEN DEST[127:64]  $\leftarrow$  DEST[127:64]

ELSE DEST [127:64]  $\leftarrow$  SRC[127:64] FI

DEST[MAXVL-1:128] (Unmodified)

### VBLENDVPD (VEX.128 encoded version)

MASK  $\leftarrow$  SRC3

IF (MASK[63] = 0) THEN DEST[63:0]  $\leftarrow$  SRC1[63:0]

ELSE DEST [63:0]  $\leftarrow$  SRC2[63:0] FI

IF (MASK[127] = 0) THEN DEST[127:64]  $\leftarrow$  SRC1[127:64]

ELSE DEST [127:64]  $\leftarrow$  SRC2[127:64] FI

DEST[MAXVL-1:128]  $\leftarrow$  0

### VBLENDVPD (VEX.256 encoded version)

MASK  $\leftarrow$  SRC3

IF (MASK[63] = 0) THEN DEST[63:0]  $\leftarrow$  SRC1[63:0]

ELSE DEST [63:0]  $\leftarrow$  SRC2[63:0] FI

IF (MASK[127] = 0) THEN DEST[127:64]  $\leftarrow$  SRC1[127:64]

ELSE DEST [127:64]  $\leftarrow$  SRC2[127:64] FI

IF (MASK[191] = 0) THEN DEST[191:128]  $\leftarrow$  SRC1[191:128]

ELSE DEST [191:128]  $\leftarrow$  SRC2[191:128] FI

IF (MASK[255] = 0) THEN DEST[255:192]  $\leftarrow$  SRC1[255:192]

ELSE DEST [255:192]  $\leftarrow$  SRC2[255:192] FI

## Intel C/C++ Compiler Intrinsic Equivalent

BLENDVPD: `__m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3);`

VBLENDVPD: `__m128 _mm_blendv_pd (__m128d a, __m128d b, __m128d mask);`

VBLENDVPD: `__m256 _mm256_blendv_pd (__m256d a, __m256d b, __m256d mask);`

## SIMD Floating-Point Exceptions

None

## Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

## BLENDVPS — Variable Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 14 /r BLENDVPS <i>xmm1</i> , <i>xmm2/m128</i> , < <i>XMM0</i> >	RM0	V/V	SSE4_1	Select packed single precision floating-point values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>XMM0</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4A /r /is4 VBLENDVPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>xmm4</i>	RVMR	V/V	AVX	Conditionally copy single-precision floating-point values from <i>xmm2</i> or <i>xmm3/m128</i> to <i>xmm1</i> , based on mask bits in the specified mask operand, <i>xmm4</i> .
VEX.NDS.256.66.0F3A.W0 4A /r /is4 VBLENDVPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>ymm4</i>	RVMR	V/V	AVX	Conditionally copy single-precision floating-point values from <i>ymm2</i> or <i>ymm3/m256</i> to <i>ymm1</i> , based on mask bits in the specified mask register, <i>ymm4</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

### Description

Conditionally copy each dword data element of single-precision floating-point value from the second source operand and the first source operand depending on mask bits defined in the mask register operand. The mask bits are the most significant bit in each dword element of the mask register.

Each quadword element of the destination operand is copied from:

- the corresponding dword element in the second source operand, if a mask bit is “1”; or
- the corresponding dword element in the first source operand, if a mask bit is “0”

The register assignment of the implicit mask operand for BLENDVPS is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPS with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed.

VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and destination operand are YMM registers. The second source operand can be a YMM register or a 256-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. VEX.W must be 0, otherwise, the instruction will #UD.

VBLENDVPS permits the mask to be any XMM or YMM register. In contrast, BLENDVPS treats XMM0 implicitly as the mask and do not support non-destructive destination operation.

## Operation

### BLENDVPS (128-bit Legacy SSE version)

MASK  $\leftarrow$  XMM0

IF (MASK[31] = 0) THEN DEST[31:0]  $\leftarrow$  DEST[31:0]  
 ELSE DEST [31:0]  $\leftarrow$  SRC[31:0] FI  
 IF (MASK[63] = 0) THEN DEST[63:32]  $\leftarrow$  DEST[63:32]  
 ELSE DEST [63:32]  $\leftarrow$  SRC[63:32] FI  
 IF (MASK[95] = 0) THEN DEST[95:64]  $\leftarrow$  DEST[95:64]  
 ELSE DEST [95:64]  $\leftarrow$  SRC[95:64] FI  
 IF (MASK[127] = 0) THEN DEST[127:96]  $\leftarrow$  DEST[127:96]  
 ELSE DEST [127:96]  $\leftarrow$  SRC[127:96] FI  
 DEST[MAXVL-1:128] (Unmodified)

### VBLENDVPS (VEX.128 encoded version)

MASK  $\leftarrow$  SRC3

IF (MASK[31] = 0) THEN DEST[31:0]  $\leftarrow$  SRC1[31:0]  
 ELSE DEST [31:0]  $\leftarrow$  SRC2[31:0] FI  
 IF (MASK[63] = 0) THEN DEST[63:32]  $\leftarrow$  SRC1[63:32]  
 ELSE DEST [63:32]  $\leftarrow$  SRC2[63:32] FI  
 IF (MASK[95] = 0) THEN DEST[95:64]  $\leftarrow$  SRC1[95:64]  
 ELSE DEST [95:64]  $\leftarrow$  SRC2[95:64] FI  
 IF (MASK[127] = 0) THEN DEST[127:96]  $\leftarrow$  SRC1[127:96]  
 ELSE DEST [127:96]  $\leftarrow$  SRC2[127:96] FI  
 DEST[MAXVL-1:128]  $\leftarrow$  0

### VBLENDVPS (VEX.256 encoded version)

MASK  $\leftarrow$  SRC3

IF (MASK[31] = 0) THEN DEST[31:0]  $\leftarrow$  SRC1[31:0]  
 ELSE DEST [31:0]  $\leftarrow$  SRC2[31:0] FI  
 IF (MASK[63] = 0) THEN DEST[63:32]  $\leftarrow$  SRC1[63:32]  
 ELSE DEST [63:32]  $\leftarrow$  SRC2[63:32] FI  
 IF (MASK[95] = 0) THEN DEST[95:64]  $\leftarrow$  SRC1[95:64]  
 ELSE DEST [95:64]  $\leftarrow$  SRC2[95:64] FI  
 IF (MASK[127] = 0) THEN DEST[127:96]  $\leftarrow$  SRC1[127:96]  
 ELSE DEST [127:96]  $\leftarrow$  SRC2[127:96] FI  
 IF (MASK[159] = 0) THEN DEST[159:128]  $\leftarrow$  SRC1[159:128]  
 ELSE DEST [159:128]  $\leftarrow$  SRC2[159:128] FI  
 IF (MASK[191] = 0) THEN DEST[191:160]  $\leftarrow$  SRC1[191:160]  
 ELSE DEST [191:160]  $\leftarrow$  SRC2[191:160] FI  
 IF (MASK[223] = 0) THEN DEST[223:192]  $\leftarrow$  SRC1[223:192]  
 ELSE DEST [223:192]  $\leftarrow$  SRC2[223:192] FI  
 IF (MASK[255] = 0) THEN DEST[255:224]  $\leftarrow$  SRC1[255:224]  
 ELSE DEST [255:224]  $\leftarrow$  SRC2[255:224] FI

## Intel C/C++ Compiler Intrinsic Equivalent

BLENDVPS: `__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3);`

VBLENDVPS: `__m128 _mm_blendv_ps (__m128 a, __m128 b, __m128 mask);`

VBLENDVPS: `__m256 _mm256_blendv_ps (__m256 a, __m256 b, __m256 mask);`

## SIMD Floating-Point Exceptions

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                      If VEX.W = 1.

**BLSI — Extract Lowest Set Isolated Bit**

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.OF38.W0 F3 /3 BLSI r32, r/m32	VM	V/V	BMI1	Extract lowest set bit from r/m32 and set that bit in r32.
VEX.NDD.LZ.OF38.W1 F3 /3 BLSI r64, r/m64	VM	V/N.E.	BMI1	Extract lowest set bit from r/m64, and set that bit in r64.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

**Description**

Extracts the lowest set bit from the source operand and set the corresponding bit in the destination register. All other bits in the destination operand are zeroed. If no bits are set in the source operand, BLSI sets all the bits in the destination to 0 and sets ZF and CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**

```
temp ← (-SRC) bitwiseAND (SRC);
SF ← temp[OperandSize - 1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 0;
ELSE
    CF ← 1;
FI
DEST ← temp;
```

**Flags Affected**

ZF and SF are updated based on the result. CF is set if the source is not zero. OF flags are cleared. AF and PF flags are undefined.

**Intel C/C++ Compiler Intrinsic Equivalent**

BLSI:        unsigned \_\_int32 \_bsi\_u32(unsigned \_\_int32 src);

BLSI:        unsigned \_\_int64 \_bsi\_u64(unsigned \_\_int64 src);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD                If VEX.W = 1.

## BLSMSK — Get Mask Up to Lowest Set Bit

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.OF38.W0 F3 /2 BLSMSK r32, r/m32	VM	V/V	BMI1	Set all lower bits in r32 to “1” starting from bit 0 to lowest set bit in r/m32.
VEX.NDD.LZ.OF38.W1 F3 /2 BLSMSK r64, r/m64	VM	V/N.E.	BMI1	Set all lower bits in r64 to “1” starting from bit 0 to lowest set bit in r/m64.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

### Description

Sets all the lower bits of the destination operand to “1” up to and including lowest set bit (=1) in the source operand. If source operand is zero, BLSMSK sets all bits of the destination operand to 1 and also sets CF to 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```
temp ← (SRC-1) XOR (SRC);
SF ← temp[OperandSize - 1];
ZF ← 0;
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

### Flags Affected

SF is updated based on the result. CF is set if the source is zero. ZF and OF flags are cleared. AF and PF flag are undefined.

### Intel C/C++ Compiler Intrinsic Equivalent

BLSMSK:     unsigned \_\_int32 \_blsmask\_u32(unsigned \_\_int32 src);

BLSMSK:     unsigned \_\_int64 \_blsmask\_u64(unsigned \_\_int64 src);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD                     If VEX.W = 1.



**BLSR — Reset Lowest Set Bit**

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.OF38.W0 F3 /1 BLSR r32, r/m32	VM	V/V	BMI1	Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32.
VEX.NDD.LZ.OF38.W1 F3 /1 BLSR r64, r/m64	VM	V/N.E.	BMI1	Reset lowest set bit of r/m64, keep all other bits of r/m64 and write result to r64.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

**Description**

Copies all bits from the source operand to the destination operand and resets (=0) the bit position in the destination operand that corresponds to the lowest set bit of the source operand. If the source operand is zero BLSR sets CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

**Operation**

```
temp ← (SRC-1) bitwiseAND ( SRC );
SF ← temp[OperandSize -1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

**Flags Affected**

ZF and SF flags are updated based on the result. CF is set if the source is zero. OF flag is cleared. AF and PF flags are undefined.

**Intel C/C++ Compiler Intrinsic Equivalent**

BLSR:        unsigned \_\_int32 \_blsr\_u32(unsigned \_\_int32 src);

BLSR:        unsigned \_\_int64 \_blsr\_u64(unsigned \_\_int64 src);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD                      If VEX.W = 1.

## BNDCL—Check Lower Bound

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1A /r BNDCL bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is lower than the lower bound in bnd.LB.
F3 0F 1A /r BNDCL bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is lower than the lower bound in bnd.LB.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	ModRM:r/m (r)	NA

### Description

Compare the address in the second operand with the lower bound in bnd. The second operand can be either a register or memory operand. If the address is lower than the lower bound in bnd.LB, it will set BNDSTATUS to 01H and signal a #BR exception.

This instruction does not cause any memory access, and does not read or write any flags.

### Operation

#### BNDCL BND, reg

```
IF reg < BND.LB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

#### BNDCL BND, mem

```
TEMP ← LEA(mem);
IF TEMP < BND.LB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
BNDCL void _bnd_chk_ptr_lbounds(const void *q)
```

### Flags Affected

None

### Protected Mode Exceptions

#BR If lower bound check fails.

#UD If the LOCK prefix is used.  
If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.  
If 67H prefix is not used and CS.D=0.  
If 67H prefix is used and CS.D=1.

### Real-Address Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

### Virtual-8086 Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
-----	--

Same exceptions as in protected mode.

**BNDU/BNDN—Check Upper Bound**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 1A /r BNDU bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 0F 1A /r BNDU bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 0F 1B /r BNDN bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).
F2 0F 1B /r BNDN bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	ModRM:r/m (r)	NA

**Description**

Compare the address in the second operand with the upper bound in bnd. The second operand can be either a register or a memory operand. If the address is higher than the upper bound in bnd.UB, it will set BNDSTATUS to 01H and signal a #BR exception.

BNDU perform 1's complement operation on the upper bound of bnd first before proceeding with address comparison. BNDN perform address comparison directly using the upper bound in bnd that is already reverted out of 1's complement form.

This instruction does not cause any memory access, and does not read or write any flags.

Effective address computation of m32/64 has identical behavior to LEA

**Operation****BNDU BND, reg**

IF reg > NOT(BND.UB) Then

BNDSTATUS ← 01H;

#BR;

FI;

**BNDU BND, mem**

TEMP ← LEA(mem);

IF TEMP > NOT(BND.UB) Then

BNDSTATUS ← 01H;

#BR;

FI;

**BNDN BND, reg**

IF reg > BND.UB Then

BNDSTATUS ← 01H;

#BR;

FI;

**BNDCN BND, mem**

```
TEMP ← LEA(mem);
IF TEMP > BND.UB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
BNDCU .void _bnd_chk_ptr_ubounds(const void *q)
```

**Flags Affected**

None

**Protected Mode Exceptions**

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 67H prefix is not used and CS.D=0. If 67H prefix is used and CS.D=1.

**Real-Address Mode Exceptions**

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

**Virtual-8086 Mode Exceptions**

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
-----	--

Same exceptions as in protected mode.

## BNDLDX—Load Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 1A /r BNDLDX bnd, mib	RM	V/V	MPX	Load the bounds stored in a bound table entry (BTE) into bnd with address translation using the base of mib and conditional on the index of mib matching the pointer value in the BTE.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	SIB.base (r): Address of pointer SIB.index(r)	NA

### Description

BNDLDX uses the linear address constructed from the base register and displacement of the SIB-addressing form of the memory operand (mib) to perform address translation to access a bound table entry and conditionally load the bounds in the BTE to the destination. The destination register is updated with the bounds in the BTE, if the content of the index register of mib matches the pointer value stored in the BTE.

If the pointer value comparison fails, the destination is updated with INIT bounds (lb = 0x0, ub = 0x0) (note: as articulated earlier, the upper bound is represented using 1's complement, therefore, the 0x0 value of upper bound allows for access to full memory).

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

### Operation

$\text{base} \leftarrow \text{mib.SIB.base} ? \text{mib.SIB.base} + \text{Disp} : 0;$

$\text{ptr\_value} \leftarrow \text{mib.SIB.index} ? \text{mib.SIB.index} : 0;$

#### Outside 64-bit mode

$\text{A\_BDE}[31:0] \leftarrow (\text{Zero\_extend32}(\text{base}[31:12] \ll 2) + (\text{BNDCFG}[31:12] \ll 12));$

$\text{A\_BT}[31:0] \leftarrow \text{LoadFrom}(\text{A\_BDE});$

IF  $\text{A\_BT}[0]$  equal 0 Then

$\text{BNDSTATUS} \leftarrow \text{A\_BDE} \mid 02\text{H};$

#BR;

FI;

$\text{A\_BTE}[31:0] \leftarrow (\text{Zero\_extend32}(\text{base}[11:2] \ll 4) + (\text{A\_BT}[31:2] \ll 2));$

$\text{Temp\_lb}[31:0] \leftarrow \text{LoadFrom}(\text{A\_BTE});$

$\text{Temp\_ub}[31:0] \leftarrow \text{LoadFrom}(\text{A\_BTE} + 4);$

$\text{Temp\_ptr}[31:0] \leftarrow \text{LoadFrom}(\text{A\_BTE} + 8);$

IF  $\text{Temp\_ptr}$  equal  $\text{ptr\_value}$  Then

$\text{BND.LB} \leftarrow \text{Temp\_lb};$

$\text{BND.UB} \leftarrow \text{Temp\_ub};$

```
ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;
```

**In 64-bit mode**

```
A_BDE[63:0] ← (Zero_extend64(base[47+MAWA:20] << 3) + (BNDCFG[63:20] << 12));1
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_BTE[63:0] ← (Zero_extend64(base[19:3] << 5) + (A_BT[63:3] << 3));
Temp_lb[63:0] ← LoadFrom(A_BTE);
Temp_ub[63:0] ← LoadFrom(A_BTE + 8);
Temp_ptr[63:0] ← LoadFrom(A_BTE + 16);
IF Temp_ptr equal ptr_value Then
    BND.LB ← Temp_lb;
    BND.UB ← Temp_ub;
ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

BNDLDX: Generated by compiler as needed.

**Flags Affected**

None

**Protected Mode Exceptions**

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 67H prefix is not used and CS.D=0. If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit. If DS register contains a NULL segment selector.
#PF(fault code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

---

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

**Virtual-8086 Mode Exceptions**

#UD	<p>If the LOCK prefix is used.</p> <p>If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.</p> <p>If 16-bit addressing is used.</p>
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#BR	If the bound directory entry is invalid.
#UD	<p>If ModRM is RIP relative.</p> <p>If the LOCK prefix is used.</p> <p>If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.</p>
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
#PF(fault code)	If a page fault occurs.



## BNDMK—Make Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1B /r BNDMK bnd, m32	RM	NE/V	MPX	Make lower and upper bounds from m32 and store them in bnd.
F3 0F 1B /r BNDMK bnd, m64	RM	V/NE	MPX	Make lower and upper bounds from m64 and store them in bnd.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	ModRM:r/m (r)	NA

### Description

Makes bounds from the second operand and stores the lower and upper bounds in the bound register bnd. The second operand must be a memory operand. The content of the base register from the memory operand is stored in the lower bound bnd.LB. The 1's complement of the effective address of m32/m64 is stored in the upper bound b.UB. Computation of m32/m64 has identical behavior to LEA.

This instruction does not cause any memory access, and does not read or write any flags.

If the instruction did not specify base register, the lower bound will be zero. The reg-reg form of this instruction retains legacy behavior (NOP).

The instruction causes an invalid-opcode exception (#UD) if executed in 64-bit mode with RIP-relative addressing.

### Operation

$BND.LB \leftarrow SRCMEM.base;$

IF 64-bit mode Then

$BND.UB \leftarrow NOT(LEA.64\_bits(SRCMEM));$

ELSE

$BND.UB \leftarrow Zero\_Extend.64\_bits(NOT(LEA.32\_bits(SRCMEM)));$

FI;

### Intel C/C++ Compiler Intrinsic Equivalent

`BNDMKvoid * _bnd_set_ptr_bounds(const void * q, size_t size);`

### Flags Affected

None

### Protected Mode Exceptions

- #UD
  - If the LOCK prefix is used.
  - If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
  - If 67H prefix is not used and CS.D=0.
  - If 67H prefix is used and CS.D=1.

### Real-Address Mode Exceptions

- #UD
  - If the LOCK prefix is used.
  - If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
  - If 16-bit addressing is used.

### Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
	If RIP-relative addressing is used.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

Same exceptions as in protected mode.

BNDMOV—Move Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 1A /r BNDMOV bnd1, bnd2/m64	RM	NE/V	MPX	Move lower and upper bound from bnd2/m64 to bound register bnd1.
66 0F 1A /r BNDMOV bnd1, bnd2/m128	RM	V/NE	MPX	Move lower and upper bound from bnd2/m128 to bound register bnd1.
66 0F 1B /r BNDMOV bnd1/m64, bnd2	MR	NE/V	MPX	Move lower and upper bound from bnd2 to bnd1/m64.
66 0F 1B /r BNDMOV bnd1/m128, bnd2	MR	V/NE	MPX	Move lower and upper bound from bnd2 to bound register bnd1/m128.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	ModRM:r/m (r)	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA

Description

BNDMOV moves a pair of lower and upper bound values from the source operand (the second operand) to the destination (the first operand). Each operation is 128-bit move. The exceptions are same as the MOV instruction. The memory format for loading/store bounds in 64-bit mode is shown in Figure 3-5.

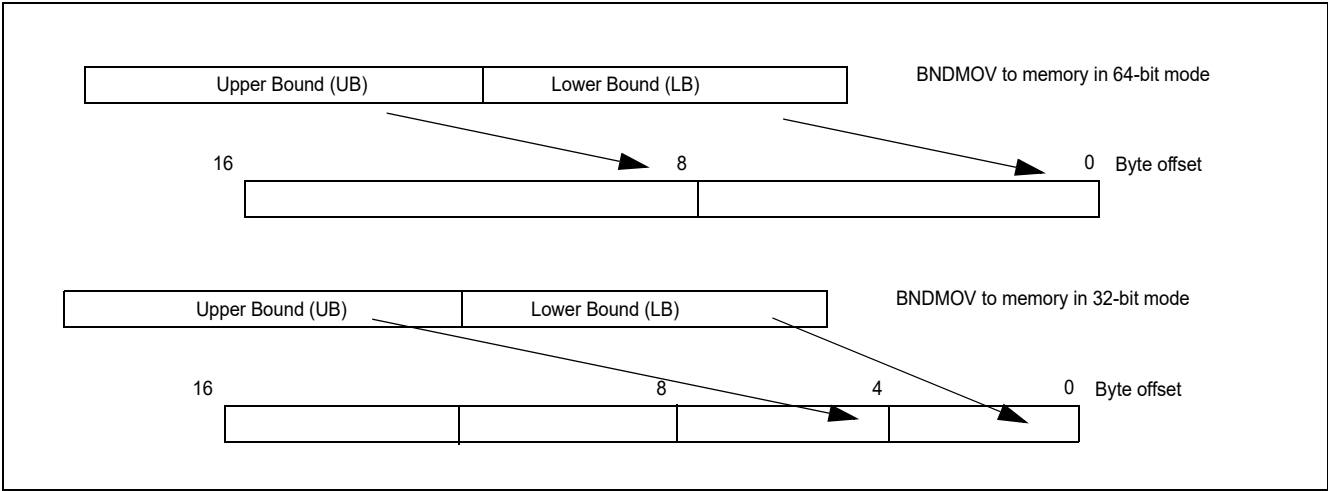


Figure 3-5. Memory Layout of BNDMOV to/from Memory

This instruction does not change flags.

Operation

**BNDMOV register to register**

DEST.LB ← SRC.LB;

DEST.UB ← SRC.UB;

**BNDMOV from memory**

```

IF 64-bit mode THEN
    DEST.LB ← LOAD_QWORD(SRC);
    DEST.UB ← LOAD_QWORD(SRC+8);
ELSE
    DEST.LB ← LOAD_DWORD_ZERO_EXT(SRC);
    DEST.UB ← LOAD_DWORD_ZERO_EXT(SRC+4);
FI;

```

**BNDMOV to memory**

```

IF 64-bit mode THEN
    DEST[63:0] ← SRC.LB;
    DEST[127:64] ← SRC.UB;
ELSE
    DEST[31:0] ← SRC.LB;
    DEST[63:32] ← SRC.UB;
FI;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
BNDMOV    void * _bnd_copy_ptr_bounds(const void *q, const void *r)
```

**Flags Affected**

None

**Protected Mode Exceptions**

#UD	<p>If the LOCK prefix is used but the destination is not a memory operand.</p> <p>If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.</p> <p>If 67H prefix is not used and CS.D=0.</p> <p>If 67H prefix is used and CS.D=1.</p>
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#GP(0)	<p>If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the destination operand points to a non-writable segment</p> <p>If the DS, ES, FS, or GS segment register contains a NULL segment selector.</p>
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#UD	<p>If the LOCK prefix is used but the destination is not a memory operand.</p> <p>If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.</p> <p>If 16-bit addressing is used.</p>
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If the memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

## BNDSTX—Store Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 1B /r BNDSTX mib, bnd	MR	V/V	MPX	Store the bounds in bnd and the pointer value in the index register of mib to a bound table entry (BTE) with address translation using the base of mib.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
MR	SIB.base (r): Address of pointer SIB.index(r)	ModRM:reg (r)	NA

### Description

BNDSTX uses the linear address constructed from the displacement and base register of the SIB-addressing form of the memory operand (mib) to perform address translation to store to a bound table entry. The bounds in the source operand bnd are written to the lower and upper bounds in the BTE. The content of the index register of mib is written to the pointer value field in the BTE.

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

### Operation

```
base ← mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;
```

### Outside 64-bit mode

```
A_BDE[31:0] ← (Zero_extend32(base[31:12] << 2) + (BNDCFG[31:12] << 12));
A_BT[31:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[31:0] ← (Zero_extend32(base[11:2] << 4) + (A_BT[31:2] << 2)); // address of Bound table entry
A_DEST[8][31:0] ← ptr_value;
A_DEST[0][31:0] ← BND.LB;
A_DEST[4][31:0] ← BND.UB;
```

**In 64-bit mode**

```

A_BDE[63:0] ← (Zero_extend64(base[47+MAWA:20] << 3) + (BNDCFG[63:20] << 12));1
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[63:0] ← (Zero_extend64(base[19:3] << 5) + (A_BT[63:3] << 3)); // address of Bound table entry
A_DEST[16][63:0] ← ptr_value;
A_DEST[0][63:0] ← BND.LB;
A_DEST[8][63:0] ← BND.UB;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
BNDSTX: _bnd_store_ptr_bounds(const void **ptr_addr, const void *ptr_val);
```

**Flags Affected**

None

**Protected Mode Exceptions**

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 67H prefix is not used and CS.D=0. If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit. If DS register contains a NULL segment selector. If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

**Virtual-8086 Mode Exceptions**

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

---

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

## 64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative. If the LOCK prefix is used. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form. If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.



**BOUND—Check Array Index Against Bounds**

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
62 /r	BOUND <i>r16</i> , <i>m16&amp;16</i>	RM	Invalid	Valid	Check if <i>r16</i> (array index) is within bounds specified by <i>m16&amp;16</i> .
62 /r	BOUND <i>r32</i> , <i>m32&amp;32</i>	RM	Invalid	Valid	Check if <i>r32</i> (array index) is within bounds specified by <i>m32&amp;32</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA	NA

**Description**

BOUND determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. When this exception is generated, the saved return instruction pointer points to the BOUND instruction.

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

**Operation**

```

IF 64bit Mode
  THEN
    #UD;
  ELSE
    IF (ArrayIndex < LowerBound OR ArrayIndex > UpperBound) THEN
      (* Below lower bound or above upper bound *)
      IF <equation for PL enabled> THEN BNDSTATUS ← 0
      #BR;
    FI;
  FI;

```

**Flags Affected**

None.

**Protected Mode Exceptions**

#BR	If the bounds test fails.
#UD	If second operand is not a memory location. If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#BR	If the bounds test fails.
#UD	If second operand is not a memory location. If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

#BR	If the bounds test fails.
#UD	If second operand is not a memory location. If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#UD	If in 64-bit mode.
-----	--------------------

## BSF—Bit Scan Forward

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF BC /r	BSF <i>r16, r/m16</i>	RM	Valid	Valid	Bit scan forward on <i>r/m16</i> .
OF BC /r	BSF <i>r32, r/m32</i>	RM	Valid	Valid	Bit scan forward on <i>r/m32</i> .
REX.W + OF BC /r	BSF <i>r64, r/m64</i>	RM	Valid	N.E.	Bit scan forward on <i>r/m64</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is 0, the content of the destination operand is undefined.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```

IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← 0;
        WHILE Bit(SRC, temp) = 0
            DO
                temp ← temp + 1;
            OD;
        DEST ← temp;
FI;

```

### Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**BSR—Bit Scan Reverse**

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF BD /r	BSR <i>r16, r/m16</i>	RM	Valid	Valid	Bit scan reverse on <i>r/m16</i> .
OF BD /r	BSR <i>r32, r/m32</i>	RM	Valid	Valid	Bit scan reverse on <i>r/m32</i> .
REX.W + OF BD /r	BSR <i>r64, r/m64</i>	RM	Valid	N.E.	Bit scan reverse on <i>r/m64</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is 0, the content of the destination operand is undefined.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

```

IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← OperandSize - 1;
        WHILE Bit(SRC, temp) = 0
            DO
                temp ← temp - 1;
            OD;
        DEST ← temp;
FI;

```

**Flags Affected**

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## BSWAP—Byte Swap

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF C8+ <i>rd</i>	BSWAP <i>r32</i>	0	Valid*	Valid	Reverses the byte order of a 32-bit register.
REX.W + OF C8+ <i>rd</i>	BSWAP <i>r64</i>	0	Valid	N.E.	Reverses the byte order of a 64-bit register.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
0	opcode + rd (r, w)	NA	NA	NA

### Description

Reverses the byte order of a 32-bit or 64-bit (destination) register. This instruction is provided for converting little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### IA-32 Architecture Legacy Compatibility

The BSWAP instruction is not supported on IA-32 processors earlier than the Intel486™ processor family. For compatibility with this instruction, software should include functionally equivalent code for execution on Intel processors earlier than the Intel486 processor family.

### Operation

TEMP ← DEST

IF 64-bit mode AND OperandSize = 64

THEN

DEST[7:0] ← TEMP[63:56];  
 DEST[15:8] ← TEMP[55:48];  
 DEST[23:16] ← TEMP[47:40];  
 DEST[31:24] ← TEMP[39:32];  
 DEST[39:32] ← TEMP[31:24];  
 DEST[47:40] ← TEMP[23:16];  
 DEST[55:48] ← TEMP[15:8];  
 DEST[63:56] ← TEMP[7:0];

ELSE

DEST[7:0] ← TEMP[31:24];  
 DEST[15:8] ← TEMP[23:16];  
 DEST[23:16] ← TEMP[15:8];  
 DEST[31:24] ← TEMP[7:0];

FI;

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

## BT—Bit Test

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF A3 /r	BT <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag.
OF A3 /r	BT <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag.
REX.W + OF A3 /r	BT <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag.
OF BA /4 <i>ib</i>	BT <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag.
OF BA /4 <i>ib</i>	BT <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag.
REX.W + OF BA /4 <i>ib</i>	BT <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r)	imm8	NA	NA

### Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset (specified by the second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode).
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

$$\text{Effective Address} + (4 * (\text{BitOffset} \text{ DIV } 32))$$

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

$$\text{Effective Address} + (2 * (\text{BitOffset} \text{ DIV } 16))$$

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

CF ← Bit(BitBase, BitOffset);



## Flags Affected

The CF flag contains the value of the selected bit. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

## Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## BTC—Bit Test and Complement

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF BB /r	BTC <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and complement.
OF BB /r	BTC <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BB /r	BTC <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and complement.
OF BA /7 <i>ib</i>	BTC <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and complement.
OF BA /7 <i>ib</i>	BTC <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + OF BA /7 <i>ib</i>	BTC <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag and complement.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

### Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

CF ← Bit(BitBase, BitOffset);

Bit(BitBase, BitOffset) ← NOT Bit(BitBase, BitOffset);

### Flags Affected

The CF flag contains the value of the selected bit before it is complemented. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## BTR—Bit Test and Reset

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF B3 /r	BTR <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and clear.
OF B3 /r	BTR <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + OF B3 /r	BTR <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and clear.
OF BA /6 <i>ib</i>	BTR <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and clear.
OF BA /6 <i>ib</i>	BTR <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + OF BA /6 <i>ib</i>	BTR <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag and clear.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

### Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

CF ← Bit(BitBase, BitOffset);  
Bit(BitBase, BitOffset) ← 0;

### Flags Affected

The CF flag contains the value of the selected bit before it is cleared. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## BTS—Bit Test and Set

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
OF AB /r	BTS <i>r/m16, r16</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
OF AB /r	BTS <i>r/m32, r32</i>	MR	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF AB /r	BTS <i>r/m64, r64</i>	MR	Valid	N.E.	Store selected bit in CF flag and set.
OF BA /5 <i>ib</i>	BTS <i>r/m16, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
OF BA /5 <i>ib</i>	BTS <i>r/m32, imm8</i>	MI	Valid	Valid	Store selected bit in CF flag and set.
REX.W + OF BA /5 <i>ib</i>	BTS <i>r/m64, imm8</i>	MI	Valid	N.E.	Store selected bit in CF flag and set.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

### Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See “BT—Bit Test” in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

CF ← Bit(BitBase, BitOffset);  
Bit(BitBase, BitOffset) ← 1;

### Flags Affected

The CF flag contains the value of the selected bit before it is set. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## BZHI — Zero High Bits Starting with Specified Bit Position

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.OF38.W0 F5 /r BZHI r32a, r/m32, r32b	RMV	V/V	BMI2	Zero bits in r/m32 starting with the position in r32b, write result to r32a.
VEX.NDS.LZ.OF38.W1 F5 /r BZHI r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Zero bits in r/m64 starting with the position in r64b, write result to r64a.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

### Description

BZHI copies the bits of the first source operand (the second operand) into the destination operand (the first operand) and clears the higher bits in the destination according to the INDEX value specified by the second source operand (the third operand). The INDEX is specified by bits 7:0 of the second source operand. The INDEX value is saturated at the value of OperandSize - 1. CF is set, if the number contained in the 8 low bits of the third operand is greater than OperandSize - 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```

N ← SRC2[7:0]
DEST ← SRC1
IF (N < OperandSize)
    DEST[OperandSize-1:N] ← 0
FI
IF (N > OperandSize - 1)
    CF ← 1
ELSE
    CF ← 0
FI

```

### Flags Affected

ZF, CF and SF flags are updated based on the result. OF flag is cleared. AF and PF flags are undefined.

### Intel C/C++ Compiler Intrinsic Equivalent

```

BZHI:    unsigned __int32 _bzhi_u32(unsigned __int32 src, unsigned __int32 index);
BZHI:    unsigned __int64 _bzhi_u64(unsigned __int64 src, unsigned __int32 index);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
 #UD If VEX.W = 1.



## CALL—Call Procedure

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
E8 <i>cw</i>	CALL <i>rel16</i>	M	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 <i>cd</i>	CALL <i>rel32</i>	M	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL <i>r/m16</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m16</i> .
FF /2	CALL <i>r/m32</i>	M	N.E.	Valid	Call near, absolute indirect, address given in <i>r/m32</i> .
FF /2	CALL <i>r/m64</i>	M	Valid	N.E.	Call near, absolute indirect, address given in <i>r/m64</i> .
9A <i>cd</i>	CALL <i>ptr16:16</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
9A <i>cp</i>	CALL <i>ptr16:32</i>	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL <i>m16:16</i>	M	Valid	Valid	Call far, absolute indirect address given in <i>m16:16</i> . In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF /3	CALL <i>m16:32</i>	M	Valid	Valid	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
REX.W + FF /3	CALL <i>m16:64</i>	M	Valid	N.E.	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

### Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls:

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 7, “Task Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for information on performing task switches with the CALL instruction.

**Near Call.** When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16*, *r/m32*, or *r/m64*). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

**Far Calls in Real-Address or Virtual-8086 Mode.** When executing a far call in real- address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a “far branch” to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

**Far Calls in Protected Mode.** When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level
- Far call to a different privilege level (inter-privilege level call)
- Task switch (far call to another task)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an optional set of parameters from the calling procedure's stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task's code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on the mechanics of a task switch.

When you execute a task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

**Mixing 16-Bit and 32-Bit Calls.** When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 21, "Mixing 16-Bit and 32-Bit Code," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information.

**Far Calls in Compatibility Mode.** When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction can not be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This is specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target

operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

**Near/(Far) Calls in 64-bit Mode.** When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

**Operation**

```

IF near call
  THEN IF near relative call
    THEN
      IF OperandSize = 64
        THEN
          tempDEST ← SignExtend(DEST); (* DEST is rel32 *)
          tempRIP ← RIP + tempDEST;
          IF stack not large enough for a 8-byte return address
            THEN #SS(0); FI;
          Push(RIP);
          RIP ← tempRIP;
        FI;
      IF OperandSize = 32
        THEN
          tempEIP ← EIP + DEST; (* DEST is rel32 *)
          IF tempEIP is not within code segment limit THEN #GP(0); FI;
          IF stack not large enough for a 4-byte return address
            THEN #SS(0); FI;
          Push(EIP);
          EIP ← tempEIP;
        FI;
      IF OperandSize = 16
        THEN
          tempEIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
          IF tempEIP is not within code segment limit THEN #GP(0); FI;
          IF stack not large enough for a 2-byte return address
            THEN #SS(0); FI;
          Push(IP);
          EIP ← tempEIP;
        FI;
    ELSE (* Near absolute call *)
      IF OperandSize = 64
        THEN
          tempRIP ← DEST; (* DEST is r/m64 *)
          IF stack not large enough for a 8-byte return address
            THEN #SS(0); FI;
          Push(RIP);
          RIP ← tempRIP;
        FI;
      IF OperandSize = 32
        THEN
          tempEIP ← DEST; (* DEST is r/m32 *)
          IF tempEIP is not within code segment limit THEN #GP(0); FI;
          IF stack not large enough for a 4-byte return address
            THEN #SS(0); FI;
          Push(EIP);
          EIP ← tempEIP;
        FI;
      IF OperandSize = 16
        THEN
          tempEIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
          IF tempEIP is not within code segment limit THEN #GP(0); FI;

```

```

        IF stack not large enough for a 2-byte return address
            THEN #SS(0); FI;
        Push(IP);
        EIP ← tempEIP;
    FI;
FI;rel/abs
FI; near

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        IF OperandSize = 32
            THEN
                IF stack not large enough for a 6-byte return address
                    THEN #SS(0); FI;
                IF DEST[31:16] is not zero THEN #GP(0); FI;
                Push(CS); (* Padded with 16 high-order bits *)
                Push(EIP);
                CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
                EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                IF stack not large enough for a 4-byte return address
                    THEN #SS(0); FI;
                Push(CS);
                Push(IP);
                CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
            FI;
        FI;
FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
    THEN
        IF segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new code segment selector); FI;
        Read type and access rights of selected segment descriptor;
        IF IA32_EFER.LMA = 0
            THEN
                IF segment type is not a conforming or nonconforming code segment, call
                    gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment or
                    64-bit call gate,
                    THEN #GP(segment selector); FI;
            FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
    FI;
FI;

```

## CONFORMING-CODE-SEGMENT:

```

    IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF DPL > CPL
        THEN #GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP ← DEST(Offset);
    IF target mode = Compatibility mode
        THEN tempEIP ← tempEIP AND 00000000_FFFFFFFFH; FI;
    IF OperandSize = 16
        THEN
            tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
    segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* Padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
    ELSE
        IF OperandSize = 16
            THEN
                Push(CS);
                Push(IP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
            ELSE (* OperandSize = 64 *)
                Push(CS); (* Padded with 48 high-order bits *)
                Push(RIP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                RIP ← tempEIP;
        FI;
    FI;
END;
```

## NONCONFORMING-CODE-SEGMENT:

```

    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) or (DPL ≠ CPL)
        THEN #GP(new code segment selector); FI;
    IF segment not present
```

```

    THEN #NP(new code segment selector); FI;
IF stack not large enough for return address
    THEN #SS(0); FI;
tempEIP ← DEST(Offset);
IF target mode = Compatibility mode
    THEN tempEIP ← tempEIP AND 00000000_FFFFFFFFH; FI;
IF OperandSize = 16
    THEN tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
segment limit)
    THEN #GP(0); FI;
IF tempEIP is non-canonical
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);
        CS ← DEST(CodeSegmentSelector);
        (* Segment descriptor information also loaded *)
        CS(RPL) ← CPL;
        EIP ← tempEIP;
    ELSE
        IF OperandSize = 16
            THEN
                Push(CS);
                Push(IP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
            ELSE (* OperandSize = 64 *)
                Push(CS); (* Padded with 48 high-order bits *)
                Push(RIP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                RIP ← tempEIP;
            FI;
        FI;
END;

```

#### CALL-GATE:

```

IF call gate (DPL < CPL) or (RPL > DPL)
    THEN #GP(call-gate selector); FI;
IF call gate not present
    THEN #NP(call-gate selector); FI;
IF call-gate code-segment selector is NULL
    THEN #GP(0); FI;
IF call-gate code-segment selector index is outside descriptor table limits
    THEN #GP(call-gate code-segment selector); FI;
Read call-gate code-segment descriptor;
IF call-gate code-segment descriptor does not indicate a code segment
or call-gate code-segment descriptor DPL > CPL
    THEN #GP(call-gate code-segment selector); FI;

```



```

IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
    THEN #GP(call-gate code-segment selector); FI;
IF call-gate code segment not present
    THEN #NP(call-gate code-segment selector); FI;
IF call-gate code segment is non-conforming and DPL < CPL
    THEN go to MORE-PRIVILEGE;
    ELSE go to SAME-PRIVILEGE;
FI;
END;

MORE-PRIVILEGE:
IF current TSS is 32-bit
    THEN
        TSSstackAddress ← (new code-segment DPL * 8) + 4;
        IF (TSSstackAddress + 5) > current TSS limit
            THEN #TS(current TSS selector); FI;
        NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
        NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
    ELSE
        IF current TSS is 16-bit
            THEN
                TSSstackAddress ← (new code-segment DPL * 4) + 2
                IF (TSSstackAddress + 3) > current TSS limit
                    THEN #TS(current TSS selector); FI;
                NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
            ELSE (* current TSS is 64-bit *)
                TSSstackAddress ← (new code-segment DPL * 8) + 4;
                IF (TSSstackAddress + 7) > current TSS limit
                    THEN #TS(current TSS selector); FI;
                NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
                NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
        FI;
    FI;
IF IA32_EFER.LMA = 0 and NewSS is NULL
    THEN #TS(NewSS); FI;
Read new stack-segment descriptor;
IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL
or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a
writable data segment)
    THEN #TS(NewSS); FI
IF IA32_EFER.LMA = 0 and new stack segment not present
    THEN #SS(NewSS); FI;
IF CallGateSize = 32
    THEN
        IF new stack does not have room for parameters plus 16 bytes
            THEN #SS(NewSS); FI;
        IF CallGate(InstructionPointer) not within new code-segment limit
            THEN #GP(0); FI;
        SS ← newSS; (* Segment descriptor information also loaded *)
        ESP ← newESP;
        CS:EIP ← CallGate(CS:InstructionPointer);
        (* Segment descriptor information also loaded *)

```

```

    Push(oldSS:oldESP); (* From calling procedure *)
    temp ← parameter count from call gate, masked to 5 bits;
    Push(parameters from calling procedure's stack, temp)
    Push(oldCS:oldEIP); (* Return address to calling procedure *)
ELSE
    IF CallGateSize = 16
        THEN
            IF new stack does not have room for parameters plus 8 bytes
                THEN #SS(NewSS); FI;
            IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
                THEN #GP(0); FI;
            SS ← newSS; (* Segment descriptor information also loaded *)
            ESP ← newESP;
            CS:IP ← CallGate(CS:InstructionPointer);
            (* Segment descriptor information also loaded *)
            Push(oldSS:oldESP); (* From calling procedure *)
            temp ← parameter count from call gate, masked to 5 bits;
            Push(parameters from calling procedure's stack, temp)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE (* CallGateSize = 64 *)
            IF pushing 32 bytes on the stack would use a non-canonical address
                THEN #SS(NewSS); FI;
            IF (CallGate(InstructionPointer) is non-canonical)
                THEN #GP(0); FI;
            SS ← NewSS; (* NewSS is NULL)
            RSP ← NewESP;
            CS:IP ← CallGate(CS:InstructionPointer);
            (* Segment descriptor information also loaded *)
            Push(oldSS:oldESP); (* From calling procedure *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        FI;
    FI;
    CPL ← CodeSegment(DPL)
    CS(RPL) ← CPL
END;

```

## SAME-PRIVILEGE:

```

    IF CallGateSize = 32
        THEN
            IF stack does not have room for 8 bytes
                THEN #SS(0); FI;
            IF CallGate(InstructionPointer) not within code segment limit
                THEN #GP(0); FI;
            CS:EIP ← CallGate(CS:EIP) (* Segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE
            IF CallGateSize = 16
                THEN
                    IF stack does not have room for 4 bytes
                        THEN #SS(0); FI;
                    IF CallGate(InstructionPointer) not within code segment limit
                        THEN #GP(0); FI;
                    CS:IP ← CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                FI;
            FI;
        FI;
    FI;

```

```

        Push(oldCS:oldIP); (* Return address to calling procedure *)
    ELSE (* CallGateSize = 64)
        IF pushing 16 bytes on the stack touches non-canonical addresses
            THEN #SS(0); FI;
        IF RIP non-canonical
            THEN #GP(0); FI;
        CS:IP ← CallGate(CS:instruction pointer);
        (* Segment descriptor information also loaded *)
        Push(oldCS:oldIP); (* Return address to calling procedure *)
    FI;
FI;
CS(RPL) ← CPL
END;

```

## TASK-GATE:

```

    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
        THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;
    IF descriptor is not a TSS segment
        THEN #GP(TSS selector); FI;
    IF TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

```

## TASK-STATE-SEGMENT:

```

    IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

```

## Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

**Protected Mode Exceptions**

#GP(0)	<p>If the target offset in destination operand is beyond the new code segment limit.</p> <p>If the segment selector in the destination operand is NULL.</p> <p>If the code segment selector in the gate is NULL.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If a code segment or gate or TSS selector index is outside descriptor table limits.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.</p> <p>If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.</p> <p>If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.</p> <p>If the segment selector from a call gate is beyond the descriptor table limits.</p> <p>If the DPL for a code-segment obtained from a call gate is greater than the CPL.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.</p> <p>If a memory operand effective address is outside the SS segment limit.</p>
#SS(selector)	<p>If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.</p> <p>If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.</p> <p>If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.</p>
#NP(selector)	If a code segment, data segment, call gate, task gate, or TSS is not present.
#TS(selector)	<p>If the new stack segment selector and ESP are beyond the end of the TSS.</p> <p>If the new stack segment selector is NULL.</p> <p>If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.</p> <p>If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.</p> <p>If the new stack segment is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the target offset is beyond the code segment limit.</p>
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the target offset is beyond the code segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

#GP(selector)	If a memory address accessed by the selector is in non-canonical space.
#GP(0)	If the target offset in the destination operand is non-canonical.

**64-Bit Mode Exceptions**

#GP(0)	If a memory address is non-canonical. If target offset in destination operand is non-canonical. If the segment selector in the destination operand is NULL. If the code segment selector in the 64-bit gate is NULL.
#GP(selector)	If code segment or 64-bit call gate is outside descriptor table limits. If code segment or 64-bit call gate overlaps non-canonical space. If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate. If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L- bit set. If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL. If the DPL for a conforming-code segment is greater than the CPL. If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate. If the upper type field of a 64-bit call gate is not 0x0. If the segment selector from a 64-bit call gate is beyond the descriptor table limits. If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL. If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear. If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
#SS(0)	If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs. If a memory operand effective address is outside the SS segment limit. If the stack address is in a non-canonical form.
#SS(selector)	If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#TS(selector)	If the load of the new RSP exceeds the limit of the TSS.
#UD	(64-bit mode only) If a far call is direct to an absolute address in memory. If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
98	CBW	Z0	Valid	Valid	AX ← sign-extend of AL.
98	CWDE	Z0	Valid	Valid	EAX ← sign-extend of AX.
REX.W + 98	CDQE	Z0	Valid	N.E.	RAX ← sign-extend of EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to doubleword) instruction copies the sign (bit 15) of the word in the AX register into the high 16 bits of the EAX register.

CBW and CWDE reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16; CWDE is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size. Others may treat these two mnemonics as synonyms (CBW/CWDE) and use the setting of the operand-size attribute to determine the size of values to be converted.

In 64-bit mode, the default operation size is the size of the destination register. Use of the REX.W prefix promotes this instruction (CDQE when promoted) to operate on 64-bit operands. In which case, CDQE copies the sign (bit 31) of the doubleword in the EAX register into the high 32 bits of RAX.

### Operation

```

IF OperandSize = 16 (* Instruction = CBW *)
    THEN
        AX ← SignExtend(AL);
    ELSE IF (OperandSize = 32, Instruction = CWDE)
        EAX ← SignExtend(AX); FI;
    ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)
        RAX ← SignExtend(EAX);
FI;

```

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

## CLAC—Clear AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CA CLAC	Z0	V/V	SMAP	Clear the AC flag in the EFLAGS register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Clears the AC flag bit in EFLAGS register. This disables any alignment checking of user-mode data accesses. If the SMAP bit is set in the CR4 register, this disallows explicit supervisor-mode data accesses to user-mode pages.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute CLAC when CPL > 0 cause #UD.

### Operation

EFLAGS.AC ← 0;

### Flags Affected

AC cleared. Other flags are unaffected.

### Protected Mode Exceptions

#UD                      If the LOCK prefix is used.  
                              If the CPL > 0.  
                              If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### Real-Address Mode Exceptions

#UD                      If the LOCK prefix is used.  
                              If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### Virtual-8086 Mode Exceptions

#UD                      The CLAC instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD                      If the LOCK prefix is used.  
                              If the CPL > 0.  
                              If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### 64-Bit Mode Exceptions

#UD                      If the LOCK prefix is used.  
                              If the CPL > 0.  
                              If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

**CLC—Clear Carry Flag**

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
F8	CLC	Z0	Valid	Valid	Clear CF flag.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Clears the CF flag in the EFLAGS register. Operation is the same in all modes.

**Operation**

$CF \leftarrow 0$ ;

**Flags Affected**

The CF flag is set to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**

#UD                      If the LOCK prefix is used.



CLD—Clear Direction Flag

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
FC	CLD	Z0	Valid	Valid	Clear DF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI). Operation is the same in all modes.

Operation

DF ← 0;

Flags Affected

The DF flag is set to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD                      If the LOCK prefix is used.

## CLFLUSH—Flush Cache Line

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
NP OF AE /7	CLFLUSH <i>m8</i>	M	Valid	Valid	Flushes cache line containing <i>m8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (CPUID.01H:EDX[bit 19]). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH/h instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH/h instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSH instruction are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, fence instructions, and executions of CLFLUSHOPT to the same cache line.<sup>1</sup> They are not ordered with respect to executions of CLFLUSHOPT to different cache lines.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSH instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSH instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSH instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```
Flush_Cache_Line(SRC);
```

### Intel C/C++ Compiler Intrinsic Equivalents

```
CLFLUSH: void _mm_clflush(void const *p)
```

1. Earlier versions of this manual specified that executions of the CLFLUSH instruction were ordered only by the MFENCE instruction. All processors implementing the CLFLUSH instruction also order it relative to the other operations enumerated above.

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0. If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0. If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
-----------------	-------------------

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0. If the LOCK prefix is used.

## CLFLUSHOPT—Flush Cache Line Optimized

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
66 0F AE /7	CLFLUSHOPT <i>m8</i>	M	Valid	Valid	Flushes cache line containing <i>m8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSHOPT is indicated by the presence of the CPUID feature flag CLFLUSHOPT (CPUID.(EAX=7,ECX=0):EBX[bit 23]). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH/h instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH/h instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSHOPT instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to the following accesses to the cache line being invalidated: writes, executions of CLFLUSH, and executions of CLFLUSHOPT. They are not ordered with respect to writes, executions of CLFLUSH, or executions of CLFLUSHOPT that access other cache lines; to enforce ordering with such an operation, software can insert an SFENCE instruction between CLFLUSHOPT and that operation.

The CLFLUSHOPT instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSHOPT instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSHOPT instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSHOPT instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSHOPT instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSHOPT instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

CLFLUSHOPT operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```
Flush_Cache_Line_Optimized(SRC);
```

### Intel C/C++ Compiler Intrinsic Equivalents

```
CLFLUSHOPT: void _mm_clflushopt(void const *p)
```

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0. If the LOCK prefix is used. If an instruction prefix F2H or F3H is used.

**Real-Address Mode Exceptions**

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0. If the LOCK prefix is used. If an instruction prefix F2H or F3H is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
-----------------	-------------------

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0. If the LOCK prefix is used. If an instruction prefix F2H or F3H is used.

## CLI — Clear Interrupt Flag

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
FA	CLI	Z0	Valid	Valid	Clear interrupt flag; interrupts disabled when interrupt flag cleared.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

In most cases, CLI clears the IF flag in the EFLAGS register and no other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no effect on the generation of exceptions and NMI interrupts.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3 and either VME mode or PVI mode is active, CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 3-7 indicates the action of the CLI instruction depending on the processor operating mode, IOPL, and CPL.

**Table 3-7. Decision Table for CLI Results**

Mode	IOPL	CLI Result
Real-address	X <sup>1</sup>	IF = 0
Protected, not PVI <sup>2</sup>	≥ CPL	IF = 0
	< CPL	#GP fault
Protected, PVI <sup>3</sup>	3	IF = 0
	0-2	VIF = 0
Virtual-8086, not VME <sup>3</sup>	3	IF = 0
	0-2	#GP fault
Virtual-8086, VME <sup>3</sup>	3	IF = 0
	0-2	VIF = 0

### NOTES:

1. X = This setting has no effect on instruction operation.

2. For this table, “protected mode” applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.

3. PVI mode and virtual-8086 mode each imply CPL = 3.

## Operation

```

IF CRO.PE = 0
  THEN IF ← 0; (* Reset Interrupt Flag *)
  ELSE
    IF IOPL ≥ CPL (* CPL = 3 if EFLAGS.VM = 1 *)
      THEN IF ← 0; (* Reset Interrupt Flag *)
      ELSE
        IF VME mode OR PVI mode
          THEN VIF ← 0; (* Reset Virtual Interrupt Flag *)
          ELSE #GP(0);
        FI;
    FI;
  FI;
FI;

```

## Flags Affected

Either the IF flag or the VIF flag is cleared to 0. Other flags are unaffected.

## Protected Mode Exceptions

#GP(0)	If CPL is greater than IOPL and PVI mode is not active. If CPL is greater than IOPL and less than 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
-----	-----------------------------

## Virtual-8086 Mode Exceptions

#GP(0)	If IOPL is less than 3 and VME mode is not active.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## CLTS—Clear Task-Switched Flag in CR0

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
0F 06	CLTS	Z0	Valid	Valid	Clears TS flag in CR0.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for more information about this flag.

CLTS operation is the same in non-64-bit modes and 64-bit mode.

See Chapter 25, “VMX Non-Root Operation,” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

CR0.TS[bit 3] ← 0;

### Flags Affected

The TS flag in CR0 register is cleared.

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0) CLTS is not recognized in virtual-8086 mode.  
 #UD If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0) If the CPL is greater than 0.  
 #UD If the LOCK prefix is used.



## CLWB—Cache Line Write Back

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F AE /6 CLWB m8	M	V/V	CLWB	Writes back modified cache line containing m8, and may retain the line in cache hierarchy in non-modified state.

### Instruction Operand Encoding<sup>1</sup>

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Writes back to memory the cache line (if modified) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line may be retained in the cache hierarchy in non-modified state. Retaining the line in the cache hierarchy is a performance optimization (treated as a hint by hardware) to reduce the possibility of cache miss on a subsequent access. Hardware may choose to retain the line at any of the levels in the cache hierarchy, and in some cases, may invalidate the line from the cache hierarchy. The source operand is a byte memory location.

The availability of CLWB instruction is indicated by the presence of the CPUID feature flag CLWB (bit 24 of the EBX register, see “CPUID — CPU Identification” in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHH instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLWB instruction is not ordered with respect to PREFETCHH instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLWB instruction that references the cache line).

CLWB instruction is ordered only by store-fencing operations. For example, software can use an SFENCE, MFENCE, XCHG, or LOCK-prefixed instructions to ensure that previous stores are included in the write-back. CLWB instruction need not be ordered by another CLWB or CLFLUSHOPT instruction. CLWB is implicitly ordered with older stores executed by the logical processor to the same address.

For usages that require only writing back modified data from cache lines to memory (do not require the line to be invalidated), and expect to subsequently access the data, software is recommended to use CLWB (with appropriate fencing) instead of CLFLUSH or CLFLUSHOPT for improved performance.

The CLWB instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load. Like a load, the CLWB instruction sets the accessed flag but not the dirty flag in the page tables.

In some implementations, the CLWB instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). CLWB instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLWB instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

### Operation

Cache\_Line\_Write\_Back(m8);

### Flags Affected

None.

---

1. ModRM.MOD != 011B

**C/C++ Compiler Intrinsic Equivalent**

CLWB void \_mm\_clwb(void const \*p);

**Protected Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.

**Real-Address Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
-----------------	-------------------

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.

**CMC—Complement Carry Flag**

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
F5	CMC	Z0	Valid	Valid	Complement CF flag.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Complements the CF flag in the EFLAGS register. CMC operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

$EFLAGS.CF[\text{bit } 0] \leftarrow \text{NOT } EFLAGS.CF[\text{bit } 0];$

**Flags Affected**

The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**

#UD If the LOCK prefix is used.

## CMOVcc—Conditional Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 47 /r	CMOVA <i>r16, r/m16</i>	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
OF 47 /r	CMOVA <i>r32, r/m32</i>	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
REX.W + OF 47 /r	CMOVA <i>r64, r/m64</i>	RM	Valid	N.E.	Move if above (CF=0 and ZF=0).
OF 43 /r	CMOVAE <i>r16, r/m16</i>	RM	Valid	Valid	Move if above or equal (CF=0).
OF 43 /r	CMOVAE <i>r32, r/m32</i>	RM	Valid	Valid	Move if above or equal (CF=0).
REX.W + OF 43 /r	CMOVAE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if above or equal (CF=0).
OF 42 /r	CMOVB <i>r16, r/m16</i>	RM	Valid	Valid	Move if below (CF=1).
OF 42 /r	CMOVB <i>r32, r/m32</i>	RM	Valid	Valid	Move if below (CF=1).
REX.W + OF 42 /r	CMOVB <i>r64, r/m64</i>	RM	Valid	N.E.	Move if below (CF=1).
OF 46 /r	CMOVBE <i>r16, r/m16</i>	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
OF 46 /r	CMOVBE <i>r32, r/m32</i>	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
REX.W + OF 46 /r	CMOVBE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if below or equal (CF=1 or ZF=1).
OF 42 /r	CMOVC <i>r16, r/m16</i>	RM	Valid	Valid	Move if carry (CF=1).
OF 42 /r	CMOVC <i>r32, r/m32</i>	RM	Valid	Valid	Move if carry (CF=1).
REX.W + OF 42 /r	CMOVC <i>r64, r/m64</i>	RM	Valid	N.E.	Move if carry (CF=1).
OF 44 /r	CMOVE <i>r16, r/m16</i>	RM	Valid	Valid	Move if equal (ZF=1).
OF 44 /r	CMOVE <i>r32, r/m32</i>	RM	Valid	Valid	Move if equal (ZF=1).
REX.W + OF 44 /r	CMOVE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if equal (ZF=1).
OF 4F /r	CMOVG <i>r16, r/m16</i>	RM	Valid	Valid	Move if greater (ZF=0 and SF=0F).
OF 4F /r	CMOVG <i>r32, r/m32</i>	RM	Valid	Valid	Move if greater (ZF=0 and SF=0F).
REX.W + OF 4F /r	CMOVG <i>r64, r/m64</i>	RM	V/N.E.	NA	Move if greater (ZF=0 and SF=0F).
OF 4D /r	CMOVGE <i>r16, r/m16</i>	RM	Valid	Valid	Move if greater or equal (SF=0F).
OF 4D /r	CMOVGE <i>r32, r/m32</i>	RM	Valid	Valid	Move if greater or equal (SF=0F).
REX.W + OF 4D /r	CMOVGE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if greater or equal (SF=0F).
OF 4C /r	CMOVL <i>r16, r/m16</i>	RM	Valid	Valid	Move if less (SF≠ 0F).
OF 4C /r	CMOVL <i>r32, r/m32</i>	RM	Valid	Valid	Move if less (SF≠ 0F).
REX.W + OF 4C /r	CMOVL <i>r64, r/m64</i>	RM	Valid	N.E.	Move if less (SF≠ 0F).
OF 4E /r	CMOVLE <i>r16, r/m16</i>	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ 0F).
OF 4E /r	CMOVLE <i>r32, r/m32</i>	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ 0F).
REX.W + OF 4E /r	CMOVLE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if less or equal (ZF=1 or SF≠ 0F).
OF 46 /r	CMOVNA <i>r16, r/m16</i>	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
OF 46 /r	CMOVNA <i>r32, r/m32</i>	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
REX.W + OF 46 /r	CMOVNA <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not above (CF=1 or ZF=1).
OF 42 /r	CMOVNAE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not above or equal (CF=1).
OF 42 /r	CMOVNAE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not above or equal (CF=1).
REX.W + OF 42 /r	CMOVNAE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not above or equal (CF=1).
OF 43 /r	CMOVNB <i>r16, r/m16</i>	RM	Valid	Valid	Move if not below (CF=0).
OF 43 /r	CMOVNB <i>r32, r/m32</i>	RM	Valid	Valid	Move if not below (CF=0).
REX.W + OF 43 /r	CMOVNB <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not below (CF=0).
OF 47 /r	CMOVNBE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 47 /r	CMOVNBE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
REX.W + OF 47 /r	CMOVNBE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not below or equal (CF=0 and ZF=0).
OF 43 /r	CMOVNC <i>r16, r/m16</i>	RM	Valid	Valid	Move if not carry (CF=0).
OF 43 /r	CMOVNC <i>r32, r/m32</i>	RM	Valid	Valid	Move if not carry (CF=0).
REX.W + OF 43 /r	CMOVNC <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not carry (CF=0).
OF 45 /r	CMOVNE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not equal (ZF=0).
OF 45 /r	CMOVNE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not equal (ZF=0).
REX.W + OF 45 /r	CMOVNE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not equal (ZF=0).
OF 4E /r	CMOVNG <i>r16, r/m16</i>	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠OF).
OF 4E /r	CMOVNG <i>r32, r/m32</i>	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠OF).
REX.W + OF 4E /r	CMOVNG <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not greater (ZF=1 or SF≠OF).
OF 4C /r	CMOVNGE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not greater or equal (SF≠OF).
OF 4C /r	CMOVNGE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not greater or equal (SF≠OF).
REX.W + OF 4C /r	CMOVNGE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not greater or equal (SF≠OF).
OF 4D /r	CMOVNL <i>r16, r/m16</i>	RM	Valid	Valid	Move if not less (SF=OF).
OF 4D /r	CMOVNL <i>r32, r/m32</i>	RM	Valid	Valid	Move if not less (SF=OF).
REX.W + OF 4D /r	CMOVNL <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not less (SF=OF).
OF 4F /r	CMOVNLE <i>r16, r/m16</i>	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
OF 4F /r	CMOVNLE <i>r32, r/m32</i>	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
REX.W + OF 4F /r	CMOVNLE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not less or equal (ZF=0 and SF=OF).
OF 41 /r	CMOVNO <i>r16, r/m16</i>	RM	Valid	Valid	Move if not overflow (OF=0).
OF 41 /r	CMOVNO <i>r32, r/m32</i>	RM	Valid	Valid	Move if not overflow (OF=0).
REX.W + OF 41 /r	CMOVNO <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not overflow (OF=0).
OF 4B /r	CMOVNP <i>r16, r/m16</i>	RM	Valid	Valid	Move if not parity (PF=0).
OF 4B /r	CMOVNP <i>r32, r/m32</i>	RM	Valid	Valid	Move if not parity (PF=0).
REX.W + OF 4B /r	CMOVNP <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not parity (PF=0).
OF 49 /r	CMOVNS <i>r16, r/m16</i>	RM	Valid	Valid	Move if not sign (SF=0).
OF 49 /r	CMOVNS <i>r32, r/m32</i>	RM	Valid	Valid	Move if not sign (SF=0).
REX.W + OF 49 /r	CMOVNS <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not sign (SF=0).
OF 45 /r	CMOVNZ <i>r16, r/m16</i>	RM	Valid	Valid	Move if not zero (ZF=0).
OF 45 /r	CMOVNZ <i>r32, r/m32</i>	RM	Valid	Valid	Move if not zero (ZF=0).
REX.W + OF 45 /r	CMOVNZ <i>r64, r/m64</i>	RM	Valid	N.E.	Move if not zero (ZF=0).
OF 40 /r	CMOVO <i>r16, r/m16</i>	RM	Valid	Valid	Move if overflow (OF=1).
OF 40 /r	CMOVO <i>r32, r/m32</i>	RM	Valid	Valid	Move if overflow (OF=1).
REX.W + OF 40 /r	CMOVO <i>r64, r/m64</i>	RM	Valid	N.E.	Move if overflow (OF=1).
OF 4A /r	CMOVPP <i>r16, r/m16</i>	RM	Valid	Valid	Move if parity (PF=1).
OF 4A /r	CMOVPP <i>r32, r/m32</i>	RM	Valid	Valid	Move if parity (PF=1).
REX.W + OF 4A /r	CMOVPP <i>r64, r/m64</i>	RM	Valid	N.E.	Move if parity (PF=1).
OF 4A /r	CMOVPE <i>r16, r/m16</i>	RM	Valid	Valid	Move if parity even (PF=1).
OF 4A /r	CMOVPE <i>r32, r/m32</i>	RM	Valid	Valid	Move if parity even (PF=1).
REX.W + OF 4A /r	CMOVPE <i>r64, r/m64</i>	RM	Valid	N.E.	Move if parity even (PF=1).

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 4B /r	CMOVPO <i>r16, r/m16</i>	RM	Valid	Valid	Move if parity odd (PF=0).
OF 4B /r	CMOVPO <i>r32, r/m32</i>	RM	Valid	Valid	Move if parity odd (PF=0).
REX.W + OF 4B /r	CMOVPO <i>r64, r/m64</i>	RM	Valid	N.E.	Move if parity odd (PF=0).
OF 48 /r	CMOVS <i>r16, r/m16</i>	RM	Valid	Valid	Move if sign (SF=1).
OF 48 /r	CMOVS <i>r32, r/m32</i>	RM	Valid	Valid	Move if sign (SF=1).
REX.W + OF 48 /r	CMOVS <i>r64, r/m64</i>	RM	Valid	N.E.	Move if sign (SF=1).
OF 44 /r	CMOVZ <i>r16, r/m16</i>	RM	Valid	Valid	Move if zero (ZF=1).
OF 44 /r	CMOVZ <i>r32, r/m32</i>	RM	Valid	Valid	Move if zero (ZF=1).
REX.W + OF 44 /r	CMOVZ <i>r64, r/m64</i>	RM	Valid	N.E.	Move if zero (ZF=1).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg ( <i>r, w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA

### Description

The CMOV<sub>cc</sub> instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOV<sub>cc</sub> instruction.

These instructions can move 16-bit, 32-bit or 64-bit values from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The condition for each CMOV<sub>cc</sub> mnemonic is given in the description column of the above table. The terms “less” and “greater” are used for comparisons of signed integers and the terms “above” and “below” are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.

The CMOV<sub>cc</sub> instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOV<sub>cc</sub> instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in this chapter).

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

temp ← SRC

IF condition TRUE

THEN

DEST ← temp;

FI;

ELSE

IF (OperandSize = 32 and IA-32e mode active)

THEN

DEST[63:32] ← 0;

FI;

FI;

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## CMP—Compare Two Operands

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
3C <i>ib</i>	CMP AL, <i>imm8</i>	I	Valid	Valid	Compare <i>imm8</i> with AL.
3D <i>iw</i>	CMP AX, <i>imm16</i>	I	Valid	Valid	Compare <i>imm16</i> with AX.
3D <i>id</i>	CMP EAX, <i>imm32</i>	I	Valid	Valid	Compare <i>imm32</i> with EAX.
REX.W + 3D <i>id</i>	CMP RAX, <i>imm32</i>	I	Valid	N.E.	Compare <i>imm32</i> sign-extended to 64-bits with RAX.
80 /7 <i>ib</i>	CMP <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with <i>r/m8</i> .
REX + 80 /7 <i>ib</i>	CMP <i>r/m8</i> <sup>*</sup> , <i>imm8</i>	MI	Valid	N.E.	Compare <i>imm8</i> with <i>r/m8</i> .
81 /7 <i>iw</i>	CMP <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Compare <i>imm16</i> with <i>r/m16</i> .
81 /7 <i>id</i>	CMP <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Compare <i>imm32</i> with <i>r/m32</i> .
REX.W + 81 /7 <i>id</i>	CMP <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Compare <i>imm32</i> sign-extended to 64-bits with <i>r/m64</i> .
83 /7 <i>ib</i>	CMP <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with <i>r/m16</i> .
83 /7 <i>ib</i>	CMP <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Compare <i>imm8</i> with <i>r/m32</i> .
REX.W + 83 /7 <i>ib</i>	CMP <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Compare <i>imm8</i> with <i>r/m64</i> .
38 /r	CMP <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Compare <i>r8</i> with <i>r/m8</i> .
REX + 38 /r	CMP <i>r/m8</i> <sup>*</sup> , <i>r8</i> <sup>*</sup>	MR	Valid	N.E.	Compare <i>r8</i> with <i>r/m8</i> .
39 /r	CMP <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Compare <i>r16</i> with <i>r/m16</i> .
39 /r	CMP <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Compare <i>r32</i> with <i>r/m32</i> .
REX.W + 39 /r	CMP <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Compare <i>r64</i> with <i>r/m64</i> .
3A /r	CMP <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Compare <i>r/m8</i> with <i>r8</i> .
REX + 3A /r	CMP <i>r8</i> <sup>*</sup> , <i>r/m8</i> <sup>*</sup>	RM	Valid	N.E.	Compare <i>r/m8</i> with <i>r8</i> .
3B /r	CMP <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Compare <i>r/m16</i> with <i>r16</i> .
3B /r	CMP <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Compare <i>r/m32</i> with <i>r32</i> .
REX.W + 3B /r	CMP <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Compare <i>r/m64</i> with <i>r64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r)	<i>imm8</i>	NA	NA
I	AL/AX/EAX/RAX (r)	<i>imm8</i>	NA	NA

### Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The condition codes used by the *Jcc*, *CMOVcc*, and *SETcc* instructions are based on the results of a CMP instruction. Appendix B, "EFLAGS Condition Codes," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, shows the relationship of the status flags and the condition codes.



In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

temp  $\leftarrow$  SRC1 – SignExtend(SRC2);

ModifyStatusFlags; (\* Modify status flags in the same manner as the SUB instruction\*)

### Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## CMPPD—Compare Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Compare packed double-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.NDS.128.66.0F.WIG C2 /r ib VCMPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed double-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.NDS.256.66.0F.WIG C2 /r ib VCMPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed double-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.NDS.128.66.0F.W1 C2 /r ib VCMPD k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed double-precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F.W1 C2 /r ib VCMPD k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed double-precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F.W1 C2 /r ib VCMPD k1 {k2}, zmm2, zmm3/m512/m64bcst{sae}, imm8	C	V/V	AVX512F	Compare packed double-precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Performs a SIMD compare of the packed double-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX or EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

**Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions**

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A > B	A < B	A = B	Unordered <sup>1</sup>	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, nonsignaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes
NGE_UQ	19H	Not-greater-than-or-equal (unordered, non-signaling)	False	True	False	True	No

**Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)**

Predicate	Imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNaN
			A > B	A < B	A = B	Unordered <sup>1</sup>	
NGT_UQ	1AH	Not-greater-than (unordered, nonsignaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignaling)	True	False	True	False	No
GT_OQ	1EH	Greater-than (ordered, nonsignaling)	True	False	False	False	No
TRUE_US	1FH	True (unordered, signaling)	True	True	True	True	Yes

**NOTES:**

1. If either operand A or B is a NaN.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX = 0". See Table 3-2. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-2. Pseudo-Op and CMPPD Implementation**

Pseudo-Op	CMPPD Implementation
CMPEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 0</i>
CMPLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 1</i>
CMPLEPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 2</i>
CMPUNORDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 3</i>
CMPNEQPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 4</i>
CMPNLTPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 5</i>
CMPNLEPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 6</i>
CMPORDPD <i>xmm1, xmm2</i>	CMPPD <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX = 1" implement the full complement of 32 predicates shown in Table 3-3, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-3, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal

syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPD instructions in a similar fashion by extending the syntax listed in Table 3-3.

**Table 3-3. Pseudo-Op and VCMPPD Implementation**

Pseudo-Op	CMPPD Implementation
VCMPPEQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0</i>
VCMPPLTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1</i>
VCMPLEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 2</i>
VCMPUNORDPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 3</i>
VCMPNEQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 4</i>
VCMPNLTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 5</i>
VCMPNLEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 6</i>
VCMPORDPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 7</i>
VCMPPEQ_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 8</i>
VCMPNGEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 9</i>
VCMPNGTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0CH</i>
VCMPGEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0DH</i>
VCMPGTPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0EH</i>
VCMPTRUEPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 0FH</i>
VCMPPEQ_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 10H</i>
VCMPPLT_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 11H</i>
VCMPLE_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 12H</i>
VCMPUNORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 13H</i>
VCMPNEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 14H</i>
VCMPNLT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 15H</i>
VCMPNLE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 16H</i>
VCMPORD_SPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 17H</i>
VCMPPEQ_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 18H</i>
VCMPNGE_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 19H</i>
VCMPNGT_UQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1AH</i>
VCMPFALSE_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1BH</i>
VCMPNEQ_OSPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1CH</i>
VCMPGE_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1DH</i>
VCMPGT_OQPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1EH</i>
VCMPTRUE_USPD <i>reg1, reg2, reg3</i>	VCMPPD <i>reg1, reg2, reg3, 1FH</i>

## Operation

CASE (COMPARISON PREDICATE) OF

```

0: OP3 ← EQ_OQ; OP5 ← EQ_OQ;
  1: OP3 ← LT_OS; OP5 ← LT_OS;
  2: OP3 ← LE_OS; OP5 ← LE_OS;
  3: OP3 ← UNORD_Q; OP5 ← UNORD_Q;
  4: OP3 ← NEQ_UQ; OP5 ← NEQ_UQ;
  5: OP3 ← NLT_US; OP5 ← NLT_US;
  6: OP3 ← NLE_US; OP5 ← NLE_US;
  7: OP3 ← ORD_Q; OP5 ← ORD_Q;
  8: OP5 ← EQ_UQ;
  9: OP5 ← NGE_US;
 10: OP5 ← NGT_US;
 11: OP5 ← FALSE_OQ;
 12: OP5 ← NEQ_OQ;
 13: OP5 ← GE_OS;
 14: OP5 ← GT_OS;
 15: OP5 ← TRUE_UQ;
 16: OP5 ← EQ_OS;
 17: OP5 ← LT_OQ;
 18: OP5 ← LE_OQ;
 19: OP5 ← UNORD_S;
 20: OP5 ← NEQ_US;
 21: OP5 ← NLT_UQ;
 22: OP5 ← NLE_UQ;
 23: OP5 ← ORD_S;
 24: OP5 ← EQ_US;
 25: OP5 ← NGE_UQ;
 26: OP5 ← NGT_UQ;
 27: OP5 ← FALSE_OS;
 28: OP5 ← NEQ_OS;
 29: OP5 ← GE_OQ;
 30: OP5 ← GT_OQ;
 31: OP5 ← TRUE_US;
  DEFAULT: Reserved;
ESAC;
```

**VCMPDP (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

CMP ← SRC1[i+63:i] OP5 SRC2[63:0]

ELSE

CMP ← SRC1[i+63:i] OP5 SRC2[i+63:i]

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VCMPDP (VEX.256 encoded version)**

CMP0 ← SRC1[63:0] OP5 SRC2[63:0];

CMP1 ← SRC1[127:64] OP5 SRC2[127:64];

CMP2 ← SRC1[191:128] OP5 SRC2[191:128];

CMP3 ← SRC1[255:192] OP5 SRC2[255:192];

IF CMP0 = TRUE

THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;

ELSE DEST[63:0] ← 0000000000000000H; FI;

IF CMP1 = TRUE

THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;

ELSE DEST[127:64] ← 0000000000000000H; FI;

IF CMP2 = TRUE

THEN DEST[191:128] ← FFFFFFFFFFFFFFFFH;

ELSE DEST[191:128] ← 0000000000000000H; FI;

IF CMP3 = TRUE

THEN DEST[255:192] ← FFFFFFFFFFFFFFFFH;

ELSE DEST[255:192] ← 0000000000000000H; FI;

DEST[MAXVL-1:256] ← 0

**VCMPDP (VEX.128 encoded version)**

CMP0 ← SRC1[63:0] OP5 SRC2[63:0];

CMP1 ← SRC1[127:64] OP5 SRC2[127:64];

IF CMP0 = TRUE

THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;

ELSE DEST[63:0] ← 0000000000000000H; FI;

IF CMP1 = TRUE

THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;

ELSE DEST[127:64] ← 0000000000000000H; FI;

DEST[MAXVL-1:128] ← 0

**CMPPD (128-bit Legacy SSE version)**

```

CMP0 ← SRC1[63:0] OP3 SRC2[63:0];
CMP1 ← SRC1[127:64] OP3 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0000000000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCMPD __mmask8 _mm512_cmp_pd_mask( __m512d a, __m512d b, int imm);
VCMPD __mmask8 _mm512_cmp_round_pd_mask( __m512d a, __m512d b, int imm, int sae);
VCMPD __mmask8 _mm512_mask_cmp_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm);
VCMPD __mmask8 _mm512_mask_cmp_round_pd_mask( __mmask8 k1, __m512d a, __m512d b, int imm, int sae);
VCMPD __mmask8 _mm256_cmp_pd_mask( __m256d a, __m256d b, int imm);
VCMPD __mmask8 _mm256_mask_cmp_pd_mask( __mmask8 k1, __m256d a, __m256d b, int imm);
VCMPD __mmask8 _mm_cmp_pd_mask( __m128d a, __m128d b, int imm);
VCMPD __mmask8 _mm_mask_cmp_pd_mask( __mmask8 k1, __m128d a, __m128d b, int imm);
VCMPD __m256 _mm256_cmp_pd(__m256d a, __m256d b, int imm)
(V)VCMPD __m128 _mm_cmp_pd(__m128d a, __m128d b, int imm)

```

**SIMD Floating-Point Exceptions**

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1.

Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.



**CMPPS—Compare Packed Single-Precision Floating-Point Values**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8	A	V/V	SSE	Compare packed single-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.NDS.128.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed single-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.NDS.256.0F.WIG C2 /r ib VCMPSS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed single-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.NDS.128.0F.W0 C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed single-precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.0F.W0 C2 /r ib VCMPSS k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed single-precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.0F.W0 C2 /r ib VCMPSS k1 {k2}, zmm2, zmm3/m512/m32bcst{sae}, imm8	C	V/V	AVX512F	Compare packed single-precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

**Description**

Performs a SIMD compare of the packed single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination XMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX = 0". See Table 3-4. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-4. Pseudo-Op and CMPPS Implementation**

Pseudo-Op	CMPPS Implementation
CMPEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 0</i>
CMPLTPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 1</i>
CMPLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 2</i>
CMPUNORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 3</i>
CMPNEQPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 4</i>
CMPNLTPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 5</i>
CMPNLEPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 6</i>
CMPORDPS <i>xmm1, xmm2</i>	CMPPS <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX = 1" implement the full complement of 32 predicates shown in Table 3-5, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-5, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 3-5.

Table 3-5. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0</i>
VCMPPLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1</i>
VCMPLEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 2</i>
VCMPUNORDPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 3</i>
VCMPNEQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 4</i>
VCMPNLTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 5</i>
VCMPNLEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 6</i>
VCMPORDPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 8</i>
VCMPNGEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 9</i>
VCMPNGTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0CH</i>
VCMPGEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0DH</i>
VCMPGTTPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0EH</i>
VCMPTRUEPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 0FH</i>
VCMPEQ_OSPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 10H</i>
VCMPLT_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 11H</i>
VCMPLE_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 12H</i>
VCMPUNORD_SPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 13H</i>
VCMPNEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 14H</i>
VCMPNLT_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 15H</i>
VCMPNLE_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 16H</i>
VCMPORD_SPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 18H</i>
VCMPNGE_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 19H</i>
VCMPNGT_UQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1AH</i>
VCMPFALSE_OSPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1BH</i>
VCMPNEQ_OSPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1CH</i>
VCMPGE_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1DH</i>
VCMPGT_OQPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1EH</i>
VCMPTRUE_USPS <i>reg1, reg2, reg3</i>	VCMPPS <i>reg1, reg2, reg3, 1FH</i>

## Operation

```

CASE (COMPARISON PREDICATE) OF
  0: OP3 ← EQ_OQ; OP5 ← EQ_OQ;
  1: OP3 ← LT_OS; OP5 ← LT_OS;
  2: OP3 ← LE_OS; OP5 ← LE_OS;
  3: OP3 ← UNORD_Q; OP5 ← UNORD_Q;
  4: OP3 ← NEQ_UQ; OP5 ← NEQ_UQ;
  5: OP3 ← NLT_US; OP5 ← NLT_US;
  6: OP3 ← NLE_US; OP5 ← NLE_US;
  7: OP3 ← ORD_Q; OP5 ← ORD_Q;
  8: OP5 ← EQ_UQ;
  9: OP5 ← NGE_US;
 10: OP5 ← NGT_US;
 11: OP5 ← FALSE_OQ;
 12: OP5 ← NEQ_OQ;
 13: OP5 ← GE_OS;
 14: OP5 ← GT_OS;
 15: OP5 ← TRUE_UQ;
 16: OP5 ← EQ_OS;
 17: OP5 ← LT_OQ;
 18: OP5 ← LE_OQ;
 19: OP5 ← UNORD_S;
 20: OP5 ← NEQ_US;
 21: OP5 ← NLT_UQ;
 22: OP5 ← NLE_UQ;
 23: OP5 ← ORD_S;
 24: OP5 ← EQ_US;
 25: OP5 ← NGE_UQ;
 26: OP5 ← NGT_UQ;
 27: OP5 ← FALSE_OS;
 28: OP5 ← NEQ_OS;
 29: OP5 ← GE_OQ;
 30: OP5 ← GT_OQ;
 31: OP5 ← TRUE_US;
  DEFAULT: Reserved
ESAC;

```

**VCMPPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

CMP ← SRC1[i+31:i] OP5 SRC2[31:0]

ELSE

CMP ← SRC1[i+31:i] OP5 SRC2[i+31:i]

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only FI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VCMPPS (VEX.256 encoded version)**

CMP0 ← SRC1[31:0] OP5 SRC2[31:0];

CMP1 ← SRC1[63:32] OP5 SRC2[63:32];

CMP2 ← SRC1[95:64] OP5 SRC2[95:64];

CMP3 ← SRC1[127:96] OP5 SRC2[127:96];

CMP4 ← SRC1[159:128] OP5 SRC2[159:128];

CMP5 ← SRC1[191:160] OP5 SRC2[191:160];

CMP6 ← SRC1[223:192] OP5 SRC2[223:192];

CMP7 ← SRC1[255:224] OP5 SRC2[255:224];

IF CMP0 = TRUE

THEN DEST[31:0] ← FFFFFFFFH;

ELSE DEST[31:0] ← 00000000H; FI;

IF CMP1 = TRUE

THEN DEST[63:32] ← FFFFFFFFH;

ELSE DEST[63:32] ← 00000000H; FI;

IF CMP2 = TRUE

THEN DEST[95:64] ← FFFFFFFFH;

ELSE DEST[95:64] ← 00000000H; FI;

IF CMP3 = TRUE

THEN DEST[127:96] ← FFFFFFFFH;

ELSE DEST[127:96] ← 00000000H; FI;

IF CMP4 = TRUE

THEN DEST[159:128] ← FFFFFFFFH;

ELSE DEST[159:128] ← 00000000H; FI;

IF CMP5 = TRUE

THEN DEST[191:160] ← FFFFFFFFH;

ELSE DEST[191:160] ← 00000000H; FI;

IF CMP6 = TRUE

THEN DEST[223:192] ← FFFFFFFFH;

ELSE DEST[223:192] ← 00000000H; FI;

IF CMP7 = TRUE

THEN DEST[255:224] ← FFFFFFFFH;

ELSE DEST[255:224] ← 00000000H; FI;

DEST[MAXVL-1:256] ← 0

**VCMPPS (VEX.128 encoded version)**

```

CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 00000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 00000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 00000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 00000000H; FI;
DEST[MAXVL-1:128] ← 0

```

**CMPPS (128-bit Legacy SSE version)**

```

CMP0 ← SRC1[31:0] OP3 SRC2[31:0];
CMP1 ← SRC1[63:32] OP3 SRC2[63:32];
CMP2 ← SRC1[95:64] OP3 SRC2[95:64];
CMP3 ← SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 00000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 00000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 00000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 00000000H; FI;
DEST[MAXVL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCMPPS __mmask16 __mm512_cmp_ps_mask( __m512 a, __m512 b, int imm);
VCMPPS __mmask16 __mm512_cmp_round_ps_mask( __m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask16 __mm512_mask_cmp_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm);
VCMPPS __mmask16 __mm512_mask_cmp_round_ps_mask( __mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPPD __mmask8 __mm256_cmp_ps_mask( __m256 a, __m256 b, int imm);
VCMPPS __mmask8 __mm256_mask_cmp_ps_mask( __mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __mmask8 __mm_cmp_ps_mask( __m128 a, __m128 b, int imm);
VCMPPS __mmask8 __mm_mask_cmp_ps_mask( __mmask8 k1, __m128 a, __m128 b, int imm);
VCMPPS __m256 __mm256_cmp_ps( __m256 a, __m256 b, int imm)
CMPPS __m128 __mm_cmp_ps( __m128 a, __m128 b, int imm)

```

**SIMD Floating-Point Exceptions**

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1.

Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
A6	CMPS <i>m8, m8</i>	Z0	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R)ESI to byte at address (R)EDI. The status flags are set accordingly.
A7	CMPS <i>m16, m16</i>	Z0	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R)ESI with word at address (R)EDI. The status flags are set accordingly.
A7	CMPS <i>m32, m32</i>	Z0	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R)ESI at dword at address (R)EDI. The status flags are set accordingly.
REX.W + A7	CMPS <i>m64, m64</i>	Z0	Valid	N.E.	Compares quadword at address (R)ESI with quadword at address (R)EDI and sets the status flags accordingly.
A6	CMPSB	Z0	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R)ESI with byte at address (R)EDI. The status flags are set accordingly.
A7	CMPSW	Z0	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R)ESI with word at address (R)EDI. The status flags are set accordingly.
A7	CMPSD	Z0	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI with dword at address ES:(E)DI; For 64-bit mode compare dword at address (R)ESI with dword at address (R)EDI. The status flags are set accordingly.
REX.W + A7	CMPSQ	Z0	Valid	N.E.	Compares quadword at address (R)ESI with quadword at address (R)EDI and sets the status flags accordingly.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Compares the byte, word, doubleword, or quadword specified with the first source operand with the byte, word, doubleword, or quadword specified with the second source operand and sets the status flags in the EFLAGS register according to the results.

Both source operands are located in memory. The address of the first source operand is read from DS:SI, DS:ESI or RSI (depending on the address-size attribute of the instruction is 16, 32, or 64, respectively). The address of the second source operand is read from ES:DI, ES:EDI or RDI (again depending on the address-size attribute of the instruction is 16, 32, or 64). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operand form is provided to allow documentation. However, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords, quadwords), but they do not have to specify the correct loca-



tion. Locations of the source operands are always specified by the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), CMPSD (doubleword comparison), or CMPSQ (quadword comparison using REX.W).

After the comparison, the (E/R)SI and (E/R)DI registers increment or decrement automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E/R)SI and (E/R)DI register increment; if the DF flag is 1, the registers decrement.) The registers increment or decrement by 1 for byte operations, by 2 for word operations, 4 for doubleword operations. If operand size is 64, RSI and RDI registers increment by 8 for quadword operations.

The CMPS, CMPSB, CMPSW, CMPSD, and CMPSQ instructions can be preceded by the REP prefix for block comparisons. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for a description of the REP prefix.

In 64-bit mode, the instruction’s default address size is 64 bits, 32 bit address size is supported using the prefix 67H. Use of the REX.W prefix promotes doubleword operation to 64 bits (see CMPSQ). See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
temp ← SRC1 - SRC2;
SetStatusFlags(temp);
```

IF (64-Bit Mode)

THEN

IF (Byte comparison)

THEN IF DF = 0

THEN

(R)E)SI ← (R)E)SI + 1;

(R)E)DI ← (R)E)DI + 1;

ELSE

(R)E)SI ← (R)E)SI - 1;

(R)E)DI ← (R)E)DI - 1;

FI;

ELSE IF (Word comparison)

THEN IF DF = 0

THEN

(R)E)SI ← (R)E)SI + 2;

(R)E)DI ← (R)E)DI + 2;

ELSE

(R)E)SI ← (R)E)SI - 2;

(R)E)DI ← (R)E)DI - 2;

FI;

ELSE IF (Doubleword comparison)

THEN IF DF = 0

THEN

(R)E)SI ← (R)E)SI + 4;

(R)E)DI ← (R)E)DI + 4;

ELSE

(R)E)SI ← (R)E)SI - 4;

(R)E)DI ← (R)E)DI - 4;

FI;

```

ELSE (* Quadword comparison *)
    THEN IF DF = 0
        (R)E)SI  $\leftarrow$  (R)E)SI + 8;
        (R)E)DI  $\leftarrow$  (R)E)DI + 8;
    ELSE
        (R)E)SI  $\leftarrow$  (R)E)SI - 8;
        (R)E)DI  $\leftarrow$  (R)E)DI - 8;
    FI;
FI;
ELSE (* Non-64-bit Mode *)
    IF (byte comparison)
        THEN IF DF = 0
            THEN
                (E)SI  $\leftarrow$  (E)SI + 1;
                (E)DI  $\leftarrow$  (E)DI + 1;
            ELSE
                (E)SI  $\leftarrow$  (E)SI - 1;
                (E)DI  $\leftarrow$  (E)DI - 1;
            FI;
        ELSE IF (Word comparison)
            THEN IF DF = 0
                (E)SI  $\leftarrow$  (E)SI + 2;
                (E)DI  $\leftarrow$  (E)DI + 2;
            ELSE
                (E)SI  $\leftarrow$  (E)SI - 2;
                (E)DI  $\leftarrow$  (E)DI - 2;
            FI;
        ELSE (* Doubleword comparison *)
            THEN IF DF = 0
                (E)SI  $\leftarrow$  (E)SI + 4;
                (E)DI  $\leftarrow$  (E)DI + 4;
            ELSE
                (E)SI  $\leftarrow$  (E)SI - 4;
                (E)DI  $\leftarrow$  (E)DI - 4;
            FI;
        FI;
    FI;
FI;

```

### Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the temporary result of the comparison.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## CMPSD—Compare Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8	A	V/V	SSE2	Compare low double-precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.LIG.F2.0F.WIG C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8	B	V/V	AVX	Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.NDS.LIG.F2.0F.W1 C2 /r ib VCMPSD k1 {k2}, xmm2, xmm3/m64{sae}, imm8	C	V/V	AVX512F	Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Compares the low double-precision floating-point values in the second source operand and the first source operand and returns the results in of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX\_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with "CPUID.1H:ECX.AVX = 0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal" predicates. These comparisons can be made either by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison)

or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.1H:ECX.AVX = 0". See Table 3-6. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-6. Pseudo-Op and CMPSD Implementation

Pseudo-Op	CMPSD Implementation
CMPEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 0</i>
CMPLTSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 1</i>
CMPLESD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 2</i>
CMPUNORDSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 3</i>
CMPNEQSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 4</i>
CMPNLTS <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 5</i>
CMPNLESD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 6</i>
CMPORDSD <i>xmm1, xmm2</i>	CMPSD <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with "CPUID.1H:ECX.AVX = 1" implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-7, where the notations of *reg1*, *reg2*, and *reg3* represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 3-7.

Table 3-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPSEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0</i>
VCMPLTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1</i>
VCMPLESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 2</i>
VCMPUNORDSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 3</i>
VCMPNEQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 4</i>
VCMPNLTS <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 5</i>
VCMPNLESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 6</i>
VCMPORDSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 7</i>
VCMPSEQ_QSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 8</i>
VCMPNGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 9</i>
VCMPNGTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_QSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0CH</i>
VCMPGESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0DH</i>

Table 3-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	VCMPSD Implementation
VCMPGTSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0EH</i>
VCMPTRUESD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 0FH</i>
VCMPSEQ_OSSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 10H</i>
VCMPLT_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 11H</i>
VCMPLE_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 12H</i>
VCMPUNORD_SSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 13H</i>
VCMPNEQ_USSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 14H</i>
VCMPNLT_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 15H</i>
VCMPNLE_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 16H</i>
VCMPORD_SSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 17H</i>
VCMPSEQ_USSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 18H</i>
VCMPNGE_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 19H</i>
VCMPNGT_UQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1AH</i>
VCMPFALSE_OSSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1BH</i>
VCMPNEQ_OSSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1CH</i>
VCMPGE_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1DH</i>
VCMPGT_OQSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1EH</i>
VCMPTTRUE_USSD <i>reg1, reg2, reg3</i>	VCMPSD <i>reg1, reg2, reg3, 1FH</i>

Software should ensure VCMPSD is encoded with VEX.L=0. Encoding VCMPSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 ← EQ\_OQ; OP5 ← EQ\_OQ;
- 1: OP3 ← LT\_OS; OP5 ← LT\_OS;
- 2: OP3 ← LE\_OS; OP5 ← LE\_OS;
- 3: OP3 ← UNORD\_Q; OP5 ← UNORD\_Q;
- 4: OP3 ← NEQ\_UQ; OP5 ← NEQ\_UQ;
- 5: OP3 ← NLT\_US; OP5 ← NLT\_US;
- 6: OP3 ← NLE\_US; OP5 ← NLE\_US;
- 7: OP3 ← ORD\_Q; OP5 ← ORD\_Q;
- 8: OP5 ← EQ\_UQ;
- 9: OP5 ← NGE\_US;
- 10: OP5 ← NGT\_US;
- 11: OP5 ← FALSE\_OQ;
- 12: OP5 ← NEQ\_OQ;
- 13: OP5 ← GE\_OS;
- 14: OP5 ← GT\_OS;
- 15: OP5 ← TRUE\_UQ;
- 16: OP5 ← EQ\_OS;
- 17: OP5 ← LT\_OQ;
- 18: OP5 ← LE\_OQ;
- 19: OP5 ← UNORD\_S;
- 20: OP5 ← NEQ\_US;
- 21: OP5 ← NLT\_UQ;

22: OP5  $\leftarrow$  NLE\_UQ;  
 23: OP5  $\leftarrow$  ORD\_S;  
 24: OP5  $\leftarrow$  EQ\_US;  
 25: OP5  $\leftarrow$  NGE\_UQ;  
 26: OP5  $\leftarrow$  NGT\_UQ;  
 27: OP5  $\leftarrow$  FALSE\_OS;  
 28: OP5  $\leftarrow$  NEQ\_OS;  
 29: OP5  $\leftarrow$  GE\_OQ;  
 30: OP5  $\leftarrow$  GT\_OQ;  
 31: OP5  $\leftarrow$  TRUE\_US;  
 DEFAULT: Reserved

ESAC;

#### VCMPSD (EVEX encoded version)

CMPO  $\leftarrow$  SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or \*no writemask\*

THEN IF CMPO = TRUE

THEN DEST[0]  $\leftarrow$  1;

ELSE DEST[0]  $\leftarrow$  0; FI;

ELSE DEST[0]  $\leftarrow$  0 ; zeroing-masking only

FI;

DEST[MAX\_KL-1:1]  $\leftarrow$  0

#### CMPSD (128-bit Legacy SSE version)

CMPO  $\leftarrow$  DEST[63:0] OP3 SRC[63:0];

IF CMPO = TRUE

THEN DEST[63:0]  $\leftarrow$  FFFFFFFF FFFFFFFFH;

ELSE DEST[63:0]  $\leftarrow$  00000000 00000000H; FI;

DEST[MAXVL-1:64] (Unmodified)

#### VCMPSD (VEX.128 encoded version)

CMPO  $\leftarrow$  SRC1[63:0] OP5 SRC2[63:0];

IF CMPO = TRUE

THEN DEST[63:0]  $\leftarrow$  FFFFFFFF FFFFFFFFH;

ELSE DEST[63:0]  $\leftarrow$  00000000 00000000H; FI;

DEST[127:64]  $\leftarrow$  SRC1[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

#### Intel C/C++ Compiler Intrinsic Equivalent

VCMPSD \_\_mmask8 \_\_mm\_cmp\_sd\_mask( \_\_m128d a, \_\_m128d b, int imm);

VCMPSD \_\_mmask8 \_\_mm\_cmp\_round\_sd\_mask( \_\_m128d a, \_\_m128d b, int imm, int sae);

VCMPSD \_\_mmask8 \_\_mm\_mask\_cmp\_sd\_mask( \_\_mmask8 k1, \_\_m128d a, \_\_m128d b, int imm);

VCMPSD \_\_mmask8 \_\_mm\_mask\_cmp\_round\_sd\_mask( \_\_mmask8 k1, \_\_m128d a, \_\_m128d b, int imm, int sae);

(V)CMPSD \_\_m128d \_\_mm\_cmp\_sd( \_\_m128d a, \_\_m128d b, const int imm)

#### SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1 Denormal.

#### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## CMPSS—Compare Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8	A	V/V	SSE	Compare low single-precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.LIG.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.NDS.LIG.F3.0F.W0 C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m32{sae}, imm8	C	V/V	AVX512F	Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Compares the low single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 128:32 of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX\_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

The unordered relationship is true when at least one of the two source operands being compared is a NaN; the ordered relationship is true when neither source operand is a NaN.

A subsequent computational instruction that uses the mask result in the destination operand as an input operand will not generate an exception, because a mask of all 0s corresponds to a floating-point value of +0.0 and a mask of all 1s corresponds to a QNaN.

Note that processors with “CPUID.1H:ECX.AVX = 0” do not implement the “greater-than”, “greater-than-or-equal”, “not-greater than”, and “not-greater-than-or-equal relations” predicates. These comparisons can be made either



by using the inverse relationship (that is, use the “not-less-than-or-equal” to make a “greater-than” comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with “CPUID.1H:ECX.AVX = 0”. See Table 3-8. Compiler should treat reserved Imm8 values as illegal syntax.

**Table 3-8. Pseudo-Op and CMPSS Implementation**

Pseudo-Op	CMPSS Implementation
CMPEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 0</i>
CMPLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 1</i>
CMPLESS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 2</i>
CMPUNORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 3</i>
CMPNEQSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 4</i>
CMPNLTSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 5</i>
CMPNLESS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 6</i>
CMPORDSS <i>xmm1, xmm2</i>	CMPSS <i>xmm1, xmm2, 7</i>

The greater-than relations that the processor does not implement require more than one instruction to emulate in software and therefore should not be implemented as pseudo-ops. (For these, the programmer should reverse the operands of the corresponding less than relations and use move instructions to ensure that the mask is moved to the correct destination register and that the source operand is left intact.)

Processors with “CPUID.1H:ECX.AVX = 1” implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPS instruction. See Table 3-9, where the notations of *reg1*, *reg2*, and *reg3* represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPS instructions in a similar fashion by extending the syntax listed in Table 3-9.

**Table 3-9. Pseudo-Op and VCMPS Implementation**

Pseudo-Op	VCMPS Implementation
VCMPEQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0</i>
VCMPLTSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1</i>
VCMPLESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 2</i>
VCMPUNORDSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 3</i>
VCMPNEQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 4</i>
VCMPNLTSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 5</i>
VCMPNLESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 6</i>
VCMPORDSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 7</i>
VCMPEQ_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 8</i>
VCMPNGESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 9</i>
VCMPNGTSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0AH</i>
VCMPFALSESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0BH</i>
VCMPNEQ_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0CH</i>
VCMPGESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0DH</i>

Table 3-9. Pseudo-Op and VCMPS Implementation

Pseudo-Op	CMPS Implementation
VCMPTSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0EH</i>
VCMPTRESS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 0FH</i>
VCMPEQ_OSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 10H</i>
VCMPLT_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 11H</i>
VCMPLT_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 12H</i>
VCMPTUNORD_SSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 13H</i>
VCMPTNEQ_USSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 14H</i>
VCMPTNLT_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 15H</i>
VCMPTNLE_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 16H</i>
VCMPTORD_SSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 17H</i>
VCMPEQ_USSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 18H</i>
VCMPTGE_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 19H</i>
VCMPTGT_UQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1AH</i>
VCMPTFALSE_OSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1BH</i>
VCMPTNEQ_OSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1CH</i>
VCMPTGE_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1DH</i>
VCMPTGT_OQSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1EH</i>
VCMPTTRUE_USSS <i>reg1, reg2, reg3</i>	VCMPS <i>reg1, reg2, reg3, 1FH</i>

Software should ensure VCMPS is encoded with VEX.L=0. Encoding VCMPS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

CASE (COMPARISON PREDICATE) OF

- 0: OP3 ← EQ\_OQ; OP5 ← EQ\_OQ;
- 1: OP3 ← LT\_OS; OP5 ← LT\_OS;
- 2: OP3 ← LE\_OS; OP5 ← LE\_OS;
- 3: OP3 ← UNORD\_Q; OP5 ← UNORD\_Q;
- 4: OP3 ← NEQ\_UQ; OP5 ← NEQ\_UQ;
- 5: OP3 ← NLT\_US; OP5 ← NLT\_US;
- 6: OP3 ← NLE\_US; OP5 ← NLE\_US;
- 7: OP3 ← ORD\_Q; OP5 ← ORD\_Q;
- 8: OP5 ← EQ\_UQ;
- 9: OP5 ← NGE\_US;
- 10: OP5 ← NGT\_US;
- 11: OP5 ← FALSE\_OQ;
- 12: OP5 ← NEQ\_OQ;
- 13: OP5 ← GE\_OS;
- 14: OP5 ← GT\_OS;
- 15: OP5 ← TRUE\_UQ;
- 16: OP5 ← EQ\_OS;
- 17: OP5 ← LT\_OQ;
- 18: OP5 ← LE\_OQ;
- 19: OP5 ← UNORD\_S;
- 20: OP5 ← NEQ\_US;
- 21: OP5 ← NLT\_UQ;

22: OP5  $\leftarrow$  NLE\_UQ;  
 23: OP5  $\leftarrow$  ORD\_S;  
 24: OP5  $\leftarrow$  EQ\_US;  
 25: OP5  $\leftarrow$  NGE\_UQ;  
 26: OP5  $\leftarrow$  NGT\_UQ;  
 27: OP5  $\leftarrow$  FALSE\_OS;  
 28: OP5  $\leftarrow$  NEQ\_OS;  
 29: OP5  $\leftarrow$  GE\_OQ;  
 30: OP5  $\leftarrow$  GT\_OQ;  
 31: OP5  $\leftarrow$  TRUE\_US;  
 DEFAULT: Reserved

ESAC;

#### **VCMPSS (EVEX encoded version)**

CMPO  $\leftarrow$  SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or \*no writemask\*  
   THEN   IF CMPO = TRUE  
         THEN DEST[0]  $\leftarrow$  1;  
         ELSE DEST[0]  $\leftarrow$  0; FI;  
   ELSE   DEST[0]  $\leftarrow$  0                   ; zeroing-masking only  
 FI;  
 DEST[MAX\_KL-1:1]  $\leftarrow$  0

#### **CMPPS (128-bit Legacy SSE version)**

CMPO  $\leftarrow$  DEST[31:0] OP3 SRC[31:0];  
 IF CMPO = TRUE  
 THEN DEST[31:0]  $\leftarrow$  FFFFFFFFH;  
 ELSE DEST[31:0]  $\leftarrow$  00000000H; FI;  
 DEST[MAXVL-1:32] (Unmodified)

#### **VCMPSS (VEX.128 encoded version)**

CMPO  $\leftarrow$  SRC1[31:0] OP5 SRC2[31:0];  
 IF CMPO = TRUE  
 THEN DEST[31:0]  $\leftarrow$  FFFFFFFFH;  
 ELSE DEST[31:0]  $\leftarrow$  00000000H; FI;  
 DEST[127:32]  $\leftarrow$  SRC1[127:32]  
 DEST[MAXVL-1:128]  $\leftarrow$  0

#### **Intel C/C++ Compiler Intrinsic Equivalent**

VCMPPS \_\_mmask8 \_\_mm\_cmp\_ss\_mask( \_\_m128 a, \_\_m128 b, int imm);  
 VCMPPS \_\_mmask8 \_\_mm\_cmp\_round\_ss\_mask( \_\_m128 a, \_\_m128 b, int imm, int sae);  
 VCMPPS \_\_mmask8 \_\_mm\_mask\_cmp\_ss\_mask( \_\_mmask8 k1, \_\_m128 a, \_\_m128 b, int imm);  
 VCMPPS \_\_mmask8 \_\_mm\_mask\_cmp\_round\_ss\_mask( \_\_mmask8 k1, \_\_m128 a, \_\_m128 b, int imm, int sae);  
 (V)CMPPS \_\_m128 \_\_mm\_cmp\_ss(\_\_m128 a, \_\_m128 b, const int imm)

#### **SIMD Floating-Point Exceptions**

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1, Denormal.

#### **Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## CMPXCHG—Compare and Exchange

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF B0/r CMPXCHG r/m8, r8	MR	Valid	Valid*	Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL.
REX + OF B0/r CMPXCHG r/m8**, r8	MR	Valid	N.E.	Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL.
OF B1/r CMPXCHG r/m16, r16	MR	Valid	Valid*	Compare AX with r/m16. If equal, ZF is set and r16 is loaded into r/m16. Else, clear ZF and load r/m16 into AX.
OF B1/r CMPXCHG r/m32, r32	MR	Valid	Valid*	Compare EAX with r/m32. If equal, ZF is set and r32 is loaded into r/m32. Else, clear ZF and load r/m32 into EAX.
REX.W + OF B1/r CMPXCHG r/m64, r64	MR	Valid	N.E.	Compare RAX with r/m64. If equal, ZF is set and r64 is loaded into r/m64. Else, clear ZF and load r/m64 into RAX.

### NOTES:

\* See the IA-32 Architecture Compatibility section below.

\*\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA

### Description

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

### Operation

(\* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word, doubleword, or quadword comparison is being performed \*)

TEMP ← DEST

IF accumulator = TEMP

THEN

ZF ← 1;

DEST ← SRC;

ELSE

ZF ← 0;

accumulator ← TEMP;

DEST ← TEMP;

FI;

## Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF C7 /1 <i>m64</i> CMPXCHG8B <i>m64</i>	M	Valid	Valid*	Compare EDX:EAX with <i>m64</i> . If equal, set ZF and load ECX:EBX into <i>m64</i> . Else, clear ZF and load <i>m64</i> into EDX:EAX.
REX.W + OF C7 /1 <i>m128</i> CMPXCHG16B <i>m128</i>	M	Valid	N.E.	Compare RDX:RAX with <i>m128</i> . If equal, set ZF and load RCX:RBX into <i>m128</i> . Else, clear ZF and load <i>m128</i> into RDX:RAX.

### NOTES:

\*See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA

### Description

Compares the 64-bit value in EDX:EAX (or 128-bit value in RDX:RAX if operand size is 128 bits) with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX (or 128-bit value in RCX:RBX) is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX (or RDX:RAX). The destination operand is an 8-byte memory location (or 16-byte memory location if operand size is 128 bits). For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value. For the RDX:RAX and RCX:RBX register pairs, RDX and RCX contain the high-order 64 bits and RAX and RBX contain the low-order 64bits of a 128-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, default operation size is 64 bits. Use of the REX.W prefix promotes operation to 128 bits. Note that CMPXCHG16B requires that the destination (memory) operand be 16-byte aligned. See the summary chart at the beginning of this section for encoding data and limits. For information on the CPUID flag that indicates CMPXCHG16B, see page 3-208.

### IA-32 Architecture Compatibility

This instruction encoding is not supported on Intel processors earlier than the Pentium processors.

## Operation

```

IF (64-Bit Mode and OperandSize = 64)
  THEN
    TEMP128 ← DEST
    IF (RDX:RAX = TEMP128)
      THEN
        ZF ← 1;
        DEST ← RCX:RBX;
      ELSE
        ZF ← 0;
        RDX:RAX ← TEMP128;
        DEST ← TEMP128;
      FI;
    FI;
  ELSE
    TEMP64 ← DEST;
    IF (EDX:EAX = TEMP64)
      THEN
        ZF ← 1;
        DEST ← ECX:EBX;
      ELSE
        ZF ← 0;
        EDX:EAX ← TEMP64;
        DEST ← TEMP64;
      FI;
    FI;
  FI;
FI;

```

## Flags Affected

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

## Protected Mode Exceptions

#UD	If the destination is not a memory operand.
#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

#UD	If the destination operand is not a memory location.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand for CMPXCHG16B is not aligned on a 16-byte boundary. If CPUID.01H:ECX.CMPXCHG16B[bit 13] = 0.
#UD	If the destination operand is not a memory location.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



**COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Compares the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory

location. The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****COMISD (all versions)**

```
RESULT ← OrderedCompare(DEST[63:0] <> SRC[63:0]) {
```

```
(* Set EFLAGS *) CASE (RESULT) OF
```

```
    UNORDERED: ZF,PF,CF ← 111;
```

```
    GREATER_THAN: ZF,PF,CF ← 000;
```

```
    LESS_THAN: ZF,PF,CF ← 001;
```

```
    EQUAL: ZF,PF,CF ← 100;
```

```
ESAC;
```

```
OF, AF, SF ← 0; }
```

### Intel C/C++ Compiler Intrinsic Equivalent

VCOMISD int \_\_mm\_comi\_round\_sd(\_\_m128d a, \_\_m128d b, int imm, int sae);  
 VCOMISD int \_\_mm\_comieq\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comilt\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comile\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comigt\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comige\_sd (\_\_m128d a, \_\_m128d b)  
 VCOMISD int \_\_mm\_comineq\_sd (\_\_m128d a, \_\_m128d b)

### SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

**COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 2F /r COMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.OF.WIG 2F /r VCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.OF.WO 2F /r VCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Compares the single-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****COMISS (all versions)**

```
RESULT ← OrderedCompare(DEST[31:0] <> SRC[31:0]) {
```

```
(* Set EFLAGS *) CASE (RESULT) OF
```

```
    UNORDERED: ZF,PF,CF ← 111;
```

```
    GREATER_THAN: ZF,PF,CF ← 000;
```

```
    LESS_THAN: ZF,PF,CF ← 001;
```

```
    EQUAL: ZF,PF,CF ← 100;
```

```
ESAC;
```

```
OF, AF, SF ← 0; }
```

### Intel C/C++ Compiler Intrinsic Equivalent

VCOMISS int \_\_mm\_comi\_round\_ss(\_\_m128 a, \_\_m128 b, int imm, int sae);  
 VCOMISS int \_\_mm\_comieq\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comilt\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comile\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comigt\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comige\_ss (\_\_m128 a, \_\_m128 b)  
 VCOMISS int \_\_mm\_comineq\_ss (\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CPUID—CPU Identification

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F A2	CPUID	Z0	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

The ID flag (bit 21) in the EFLAGS register indicates support for the CPUID instruction. If a software procedure can set and clear this flag, the processor executing the procedure supports the CPUID instruction. This instruction operates the same in non-64-bit modes and 64-bit mode.

CPUID returns processor identification and feature information in the EAX, EBX, ECX, and EDX registers.<sup>1</sup> The instruction's output is dependent on the contents of the EAX register upon execution (in some cases, ECX as well). For example, the following pseudocode loads EAX with 00H and causes CPUID to return a Maximum Return Value and the Vendor Identification String in the appropriate registers:

```
MOV EAX, 00H
CPUID
```

Table 3-8 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using the Intel Core i7 processor, the following is true:

```
CPUID.EAX = 05H (* Returns MONITOR/MWAIT leaf. *)
CPUID.EAX = 0AH (* Returns Architectural Performance Monitoring leaf. *)
CPUID.EAX = 0BH (* Returns Extended Topology Enumeration leaf. *)
CPUID.EAX = 0CH (* INVALID: Returns the same information as CPUID.EAX = 0BH. *)
CPUID.EAX = 80000008H (* Returns linear/physical address size data. *)
CPUID.EAX = 8000000AH (* INVALID: Returns same information as CPUID.EAX = 0BH. *)
```

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

When CPUID returns the highest basic leaf information as a result of an invalid input EAX value, any dependence on input ECX value in the basic leaf is honored.

CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.

### See also:

"Serializing Instructions" in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

"Caching Translation Information" in Chapter 4, "Paging," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

1. On Intel 64 processors, CPUID clears the high 32 bits of the RAX/RBX/RCX/RDX registers in all modes.

**Table 3-8. Information Returned by CPUID Instruction**

Initial EAX Value	Information Provided about the Processor	
Basic CPUID Information		
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information. "Genu" "ntel" "inel"
01H	EAX EBX ECX EDX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6). Bits 07 - 00: Brand Index. Bits 15 - 08: CLFLUSH line size (Value * 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23 - 16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31 - 24: Initial APIC ID. Feature Information (see Figure 3-7 and Table 3-10). Feature Information (see Figure 3-8 and Table 3-11). <b>NOTES:</b> * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1.
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 3-12). Cache and TLB Information. Cache and TLB Information. Cache and TLB Information.
03H	EAX EBX ECX EDX	Reserved. Reserved. Bits 00 - 31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) Bits 32 - 63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) <b>NOTES:</b> Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.
CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.		
Deterministic Cache Parameters Leaf		
04H	EAX	<b>NOTES:</b> Leaf 04H output depends on the initial value in ECX.* See also: "INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level" on page 216. Bits 04 - 00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
		<p>Bits 07 - 05: Cache Level (starts at 1).            Bit 08: Self Initializing cache level (does not need SW initialization).            Bit 09: Fully Associative cache.</p> <p>Bits 13 - 10: Reserved.            Bits 25 - 14: Maximum number of addressable IDs for logical processors sharing this cache**, ***.            Bits 31 - 26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****.</p> <p>EBX Bits 11 - 00: L = System Coherency Line Size**.            Bits 21 - 12: P = Physical Line partitions**.            Bits 31 - 22: W = Ways of associativity**.</p> <p>ECX Bits 31-00: S = Number of Sets**.</p> <p>EDX Bit 00: Write-Back Invalidate/Invalidate.            0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache.            1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache.</p> <p>Bit 01: Cache Inclusiveness.            0 = Cache is not inclusive of lower cache levels.            1 = Cache is inclusive of lower cache levels.</p> <p>Bit 02: Complex Cache Indexing.            0 = Direct mapped cache.            1 = A complex function is used to index the cache, potentially using all address bits.</p> <p>Bits 31 - 03: Reserved = 0.</p> <p><b>NOTES:</b></p> <p>* If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0.</p> <p>** Add one to the return value to get the result.</p> <p>***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache.</p> <p>**** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID.</p> <p>***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.</p>
	<i>MONITOR/MWAIT Leaf</i>	
05H	EAX  EBX  ECX	<p>Bits 15 - 00: Smallest monitor-line size in bytes (default is processor's monitor granularity).            Bits 31 - 16: Reserved = 0.</p> <p>Bits 15 - 00: Largest monitor-line size in bytes (default is processor's monitor granularity).            Bits 31 - 16: Reserved = 0.</p> <p>Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported.            Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled.            Bits 31 - 02: Reserved.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	EDX	<p>Bits 03 - 00: Number of C0* sub C-states supported using MWAIT.            Bits 07 - 04: Number of C1* sub C-states supported using MWAIT.            Bits 11 - 08: Number of C2* sub C-states supported using MWAIT.            Bits 15 - 12: Number of C3* sub C-states supported using MWAIT.            Bits 19 - 16: Number of C4* sub C-states supported using MWAIT.            Bits 23 - 20: Number of C5* sub C-states supported using MWAIT.            Bits 27 - 24: Number of C6* sub C-states supported using MWAIT.            Bits 31 - 28: Number of C7* sub C-states supported using MWAIT.</p> <p><b>NOTE:</b>            * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.</p>
<i>Thermal and Power Management Leaf</i>		
06H	EAX	<p>Bit 00: Digital temperature sensor is supported if set.            Bit 01: Intel Turbo Boost Technology available (see description of IA32_MISC_ENABLE[38]).            Bit 02: ARAT. APIC-Timer-always-running feature is supported if set.            Bit 03: Reserved.            Bit 04: PLN. Power limit notification controls are supported if set.            Bit 05: ECMD. Clock modulation duty cycle extension is supported if set.            Bit 06: PTM. Package thermal management is supported if set.            Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set.            Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set.            Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set.            Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set.            Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set.            Bit 12: Reserved.            Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set.            Bit 14: Intel® Turbo Boost Max Technology 3.0 available.            Bits 31 - 15: Reserved.</p>
	EBX	<p>Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor.            Bits 31 - 04: Reserved.</p>
	ECX	<p>Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency.            Bits 02 - 01: Reserved = 0.            Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1BOH).            Bits 31 - 04: Reserved = 0.</p>
	EDX	Reserved = 0.



**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)		
07H	Sub-leaf 0 (Input ECX = 0). *	
EAX	Bits 31 - 00: Reports the maximum input value for supported leaf 7 sub-leaves.	
EBX	Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bit 01: IA32_TSC_ADJUST MSR is supported if 1. Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1. Bit 03: BMI1. Bit 04: HLE. Bit 05: AVX2. Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1. Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1. Bit 08: BMI2. Bit 09: Supports Enhanced REP MOVSB/STOSB if 1. Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers. Bit 11: RTM. Bit 12: RDT-M. Supports Intel® Resource Director Technology (Intel® RDT) Monitoring capability if 1. Bit 13: Deprecates FPU CS and FPU DS values if 1. Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1. Bit 15: RDT-A. Supports Intel® Resource Director Technology (Intel® RDT) Allocation capability if 1. Bit 16: AVX512F. Bit 17: AVX512DQ. Bit 18: RDSEED. Bit 19: ADX. Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1. Bit 21: AVX512_IFMA. Bit 22: Reserved. Bit 23: CLFLUSHOPT. Bit 24: CLWB. Bit 25: Intel Processor Trace. Bits 27 - 26: Reserved. Bit 28: AVX512CD. Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1. Bit 30: AVX512BW. Bit 31: AVX512VL.	
ECX	Bit 00: PREFETCHWT1. Bit 01: AVX512_VBMI. Bit 02: UMIP. Supports user-mode instruction prevention if 1. Bit 03: PKU. Supports protection keys for user-mode pages if 1. Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions). Bits 16 - 5: Reserved. Bits 21 - 17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode. Bit 22: RDPID. Supports Read Processor ID if 1. Bits 29 - 23: Reserved. Bit 30: SGX_LC. Supports SGX Launch Configuration if 1. Bit 31: Reserved.	

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	EDX	Reserved.
		<b>NOTE:</b> * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.
<i>Direct Cache Access Information Leaf</i>		
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H).
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
<i>Architectural Performance Monitoring Leaf</i>		
0AH	EAX	Bits 07 - 00: Version ID of architectural performance monitoring. Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor. Bits 23 - 16: Bit width of general-purpose, performance monitoring counter. Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events.
	EBX	Bit 00: Core cycle event not available if 1. Bit 01: Instruction retired event not available if 1. Bit 02: Reference cycles event not available if 1. Bit 03: Last-level cache reference event not available if 1. Bit 04: Last-level cache misses event not available if 1. Bit 05: Branch instruction retired event not available if 1. Bit 06: Branch mispredict retired event not available if 1. Bits 31 - 07: Reserved = 0.
	ECX	Reserved = 0.
	EDX	Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1). Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID > 1). Bits 14 - 13: Reserved = 0. Bit 15: AnyThread deprecation. Bits 31 - 16: Reserved = 0.
<i>Extended Topology Enumeration Leaf</i>		
0BH		<b>NOTES:</b> Most of Leaf 0BH output depends on the initial value in ECX. The EDX output of leaf 0BH is always valid and does not vary with input value in ECX. Output value in ECX[7:0] always equals input value in ECX[7:0]. For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0. If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX > n also return 0 in ECX[15:8].
	EAX	Bits 04 - 00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31 - 05: Reserved.
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31 - 16: Reserved.

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	ECX	Bits 07 - 00: Level number. Same value in ECX input. Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31 - 00: x2APIC ID the current logical processor.  <b>NOTES:</b> * Software should use this field (EAX[4:0]) to enumerate processor topology of the system. ** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. *** The value of the “level type” field is not related to level numbers in any way, higher “level type” values do not mean higher levels. Level type field has the following encoding: 0: Invalid. 1: SMT. 2: Core. 3-255: Reserved.
<i>Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)</i>		
0DH		<b>NOTES:</b> Leaf 0DH main leaf (ECX = 0).
	EAX	Bits 31 - 00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1. Bit 00: x87 state. Bit 01: SSE state. Bit 02: AVX state. Bits 04 - 03: MPX state. Bits 07 - 05: AVX-512 state. Bit 08: Used for IA32_XSS. Bit 09: PKRU state. Bits 31 - 10: Reserved.
	EBX	Bits 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
	ECX	Bit 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCR0.
	EDX	Bit 31 - 00: Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.
<i>Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)</i>		
0DH	EAX	Bit 00: XSAVEOPT is available. Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set. Bit 02: Supports XGETBV with ECX = 1 if set. Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set. Bits 31 - 04: Reserved.
	EBX	Bits 31 - 00: The size in bytes of the XSAVE area containing all states enabled by XCR0   IA32_XSS.

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	ECX	Bits 31 - 00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1. Bits 07 - 00: Used for XCRO. Bit 08: PT state. Bit 09: Used for XCRO. Bits 31 - 10: Reserved.
	EDX	Bits 31 - 00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.
<i>Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n &gt; 1)</i>		
0DH		<b>NOTES:</b> Leaf 0DH output depends on the initial value in ECX. Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCRO register or the IA32_XSS MSR. * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n ( $0 \leq n \leq 31$ ) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n ( $32 \leq n \leq 63$ ) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].
	EAX	Bits 31 - 0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, <i>n</i> .
	EBX	Bits 31 - 0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, <i>n</i> , does not map to a valid bit in the XCRO register*.
	ECX	Bit 00 is set if the bit <i>n</i> (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit <i>n</i> is instead supported in XCRO. Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component). Bits 31 - 02 are reserved. This field reports 0 if the sub-leaf index, <i>n</i> , is invalid*.
	EDX	This field reports 0 if the sub-leaf index, <i>n</i> , is invalid*; otherwise it is reserved.
<i>Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)</i>		
0FH		<b>NOTES:</b> Leaf 0FH output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.
	EAX	Reserved.
	EBX	Bits 31 - 00: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	Bit 00: Reserved. Bit 01: Supports L3 Cache Intel RDT Monitoring if 1. Bits 31 - 02: Reserved.
<i>L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)</i>		
0FH		<b>NOTES:</b> Leaf 0FH output depends on the initial value in ECX.
	EAX	Reserved.
	EBX	Bits 31 - 00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	ECX	Maximum range (zero-based) of RMID of this resource type.
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bit 01: Supports L3 Total Bandwidth monitoring if 1. Bit 02: Supports L3 Local Bandwidth monitoring if 1. Bits 31 - 03: Reserved.
<i>Intel Resource Director Technology (Intel RDT) Allocation Enumeration Sub-leaf (EAX = 10H, ECX = 0)</i>		
10H		<b>NOTES:</b> Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX.
	EAX	Reserved.
	EBX	Bit 00: Reserved. Bit 01: Supports L3 Cache Allocation Technology if 1. Bit 02: Supports L2 Cache Allocation Technology if 1. Bit 03: Supports Memory Bandwidth Allocation if 1. Bits 31 - 04: Reserved.
	ECX	Reserved.
	EDX	Reserved.
<i>L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 1)</i>		
10H		<b>NOTES:</b> Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved.
	EBX	Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.
	ECX	Bits 01 - 00: Reserved. Bit 02: Code and Data Prioritization Technology supported if 1. Bits 31 - 03: Reserved.
	EDX	Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
<i>L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 2)</i>		
10H		<b>NOTES:</b> Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved.
	EBX	Bits 31 - 00: Bit-granular map of isolation/contention of allocation units.
	ECX	Bits 31 - 00: Reserved.
	EDX	Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
<i>Memory Bandwidth Allocation Enumeration Sub-leaf (EAX = 10H, ECX = ResID = 3)</i>		
10H		<b>NOTES:</b> Leaf 10H output depends on the initial value in ECX.
	EAX	Bits 11 - 00: Reports the maximum MBA throttling value supported for the corresponding ResID using minus-one notation. Bits 31 - 12: Reserved.

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	EBX	Bits 31 - 00: Reserved.
	ECX	Bits 01 - 00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31 - 03: Reserved.
	EDX	Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
<i>Intel SGX Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)</i>		
12H		<b>NOTES:</b> Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions. Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions. Bit 31 - 02: Reserved.
	EBX	Bit 31 - 00: MISCSELECT. Bit vector of supported extended SGX features.
	ECX	Bit 31 - 00: Reserved.
	EDX	Bit 07 - 00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is $2^{(EDX[7:0])}$ . Bit 15 - 08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is $2^{(EDX[15:8])}$ . Bits 31 - 16: Reserved.
<i>Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)</i>		
12H		<b>NOTES:</b> Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.
	EBX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.
	ECX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.
	EDX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.
<i>Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)</i>		
12H		<b>NOTES:</b> Leaf 12H sub-leaf 2 or higher (ECX >= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1. For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EBX,EAX[31:4] depends on the sub-leaf type listed below.
	EAX	Bit 03 - 00: Sub-leaf Type 0000b: Indicates this sub-leaf is invalid. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other type encodings are reserved.
	Type	0000b. This sub-leaf is invalid. EDX:ECX:EBX:EAX return 0.

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	Type	<p>0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows.</p> <p>EAX[11:04]: Reserved (enumerate 0). EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section.</p> <p>EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section. EBX[31:20]: Reserved.</p> <p>ECX[03:00]: EPC section property encoding defined as follows: If EAX[3:0] 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If EAX[3:0] 0001b, then this section has confidentiality and integrity protection. All other encodings are reserved. ECX[11:04]: Reserved (enumerate 0). ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory.</p> <p>EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[31:20]: Reserved.</p>
<i>Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)</i>		
14H		<p><b>NOTES:</b> Leaf 14H main leaf (ECX = 0).</p> <p>EAX      Bits 31 - 00: Reports the maximum sub-leaf supported in leaf 14H.</p> <p>EBX      Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode. Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset. Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets. Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. Bit 31 - 06: Reserved.</p> <p>ECX      Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. Bit 02: If 1, indicates support of Single-Range Output scheme. Bit 03: If 1, indicates support of output to Trace Transport subsystem. Bit 30 - 04: Reserved. Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.</p> <p>EDX      Bits 31 - 00: Reserved.</p>
<i>Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)</i>		
14H	EAX	<p>Bits 02 - 00: Number of configurable Address Ranges for filtering. Bits 15 - 03: Reserved. Bits 31 - 16: Bitmap of supported MTC period encodings.</p>
	EBX	<p>Bits 15 - 00: Bitmap of supported Cycle Threshold value encodings. Bit 31 - 16: Bitmap of supported Configurable PSB frequency encodings.</p>
	ECX	<p>Bits 31 - 00: Reserved.</p>

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 31 - 00: Reserved.
<i>Time Stamp Counter and Nominal Core Crystal Clock Information Leaf</i>		
15H		<p><b>NOTES:</b></p> <p>If EBX[31:0] is 0, the TSC/"core crystal clock" ratio is not enumerated.</p> <p>EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency.</p> <p>If ECX is 0, the nominal core crystal clock frequency is not enumerated.</p> <p>"TSC frequency" = "core crystal clock frequency" * EBX/EAX.</p> <p>The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.</p> <p>EAX      Bits 31 - 00: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.</p> <p>EBX      Bits 31 - 00: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.</p> <p>ECX      Bits 31 - 00: An unsigned integer which is the nominal frequency of the core crystal clock in Hz.</p> <p>EDX      Bits 31 - 00: Reserved = 0.</p>
<i>Processor Frequency Information Leaf</i>		
16H	EAX	Bits 15 - 00: Processor Base Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	EBX	Bits 15 - 00: Maximum Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	ECX	Bits 15 - 00: Bus (Reference) Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	EDX	Reserved.
		<p><b>NOTES:</b></p> <p>* Data is returned from this interface in accordance with the processor's specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces.</p> <p>While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.</p>
<i>System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)</i>		
17H		<p><b>NOTES:</b></p> <p>Leaf 17H main leaf (ECX = 0).</p> <p>Leaf 17H output depends on the initial value in ECX.</p> <p>Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String.</p> <p>Leaf 17H is valid if MaxSOCID_Index &gt;= 3.</p> <p>Leaf 17H sub-leaves 4 and above are reserved.</p> <p>EAX      Bits 31 - 00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.</p> <p>EBX      Bits 15 - 00: SOC Vendor ID. Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel. Bits 31 - 17: Reserved = 0.</p> <p>ECX      Bits 31 - 00: Project ID. A unique number an SOC vendor assigns to its SOC projects.</p> <p>EDX      Bits 31 - 00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.</p>



**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)		
17H	EAX EBX ECX EDX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string. Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string. Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string. Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.  <b>NOTES:</b> Leaf 17H output depends on the initial value in ECX. SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H. The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.
System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX > MaxSOCID_Index)		
17H	EAX EBX ECX EDX	<b>NOTES:</b> Leaf 17H output depends on the initial value in ECX.  Bits 31 - 00: Reserved = 0. Bits 31 - 00: Reserved = 0. Bits 31 - 00: Reserved = 0. Bits 31 - 00: Reserved = 0.
Deterministic Address Translation Parameters Main Leaf (EAX = 18H, ECX = 0)		
18H	EAX EBX  	

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 04 - 00: Translation cache type field. 00000b: Null (indicates this sub-leaf is not valid). 00001b: Data TLB. 00010b: Instruction TLB. 00011b: Unified TLB*. All other encodings are reserved. Bits 07 - 05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13 - 09: Reserved. Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache** Bits 31 - 26: Reserved.
<i>Deterministic Address Translation Parameters Sub-leaf (EAX = 18H, ECX ≥ 1)</i>		
18H		<b>NOTES:</b> Each sub-leaf enumerates a different address translation structure. If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure. * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch). Please see the <i>Intel® 64 and IA-32 Architectures Optimization Reference Manual</i> for details of a particular product. ** Add one to the return value to get the result.
	EAX	Bits 31 - 00: Reserved.
	EBX	Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07 - 04: Reserved. Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15 - 11: Reserved. Bits 31 - 16: W = Ways of associativity.
	ECX	Bits 31 - 00: S = Number of Sets.
	EDX	Bits 04 - 00: Translation cache type field. 0000b: Null (indicates this sub-leaf is not valid). 0001b: Data TLB. 0010b: Instruction TLB. 0011b: Unified TLB*. All other encodings are reserved. Bits 07 - 05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13 - 09: Reserved. Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache** Bits 31 - 26: Reserved.
<i>Unimplemented CPUID Leaf Functions</i>		
40000000H - 4FFFFFFFH		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
Extended Function CPUID Information		
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information.
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved.
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode. Bits 04 - 01: Reserved. Bit 05: LZCNT. Bits 07 - 06: Reserved. Bit 08: PREFETCHW. Bits 31 - 09: Reserved.
	EDX	Bits 10 - 00: Reserved. Bit 11: SYSCALL/SYSRET available in 64-bit mode. Bits 19 - 12: Reserved = 0. Bit 20: Execute Disable Bit available. Bits 25 - 21: Reserved = 0. Bit 26: 1-GByte pages are available if 1. Bit 27: RDTSCP and IA32_TSC_AUX are available if 1. Bit 28: Reserved = 0. Bit 29: Intel® 64 Architecture available if 1. Bits 31 - 30: Reserved = 0.
80000002H	EAX	Processor Brand String.
	EBX	Processor Brand String Continued.
	ECX	Processor Brand String Continued.
	EDX	Processor Brand String Continued.
80000003H	EAX	Processor Brand String Continued.
	EBX	Processor Brand String Continued.
	ECX	Processor Brand String Continued.
	EDX	Processor Brand String Continued.
80000004H	EAX	Processor Brand String Continued.
	EBX	Processor Brand String Continued.
	ECX	Processor Brand String Continued.
	EDX	Processor Brand String Continued.
80000005H	EAX	Reserved = 0.
	EBX	Reserved = 0.
	ECX	Reserved = 0.
	EDX	Reserved = 0.
80000006H	EAX	Reserved = 0.
	EBX	Reserved = 0.
	ECX	Bits 07 - 00: Cache Line size in bytes. Bits 11 - 08: Reserved. Bits 15 - 12: L2 Associativity field *. Bits 31 - 16: Cache size in 1K units.
	EDX	Reserved = 0.

**Table 3-8. Information Returned by CPUID Instruction (Contd.)**

Initial EAX Value	Information Provided about the Processor	
		<b>NOTES:</b> * L2 associativity field encodings: 00H - Disabled. 01H - Direct mapped. 02H - 2-way. 04H - 4-way. 06H - 8-way. 08H - 16-way. 0FH - Fully associative.
80000007H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Bits 07 - 00: Reserved = 0. Bit 08: Invariant TSC available if 1. Bits 31 - 09: Reserved = 0.
80000008H	EAX  EBX ECX EDX	Linear/Physical Address size. Bits 07 - 00: #Physical Address Bits*. Bits 15 - 08: #Linear Address Bits. Bits 31 - 16: Reserved = 0. Reserved = 0. Reserved = 0. Reserved = 0.  <b>NOTES:</b> * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

### INPUT EAX = 0: Returns CPUID's Highest Value for Basic Processor Information and the Vendor Identification String

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register and is processor specific.

A vendor identification string is also returned in EBX, EDX, and ECX. For Intel processors, the string is "GenuineIntel" and is expressed:

EBX ← 756e6547h (\* "Genu", with G in the low eight bits of BL \*)

EDX ← 49656e69h (\* "inel", with i in the low eight bits of DL \*)

ECX ← 6c65746eh (\* "ntel", with n in the low eight bits of CL \*)

### INPUT EAX = 80000000H: Returns CPUID's Highest Value for Extended Processor Information

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

#### IA32\_BIOS\_SIGN\_ID Returns Microcode Update Signature

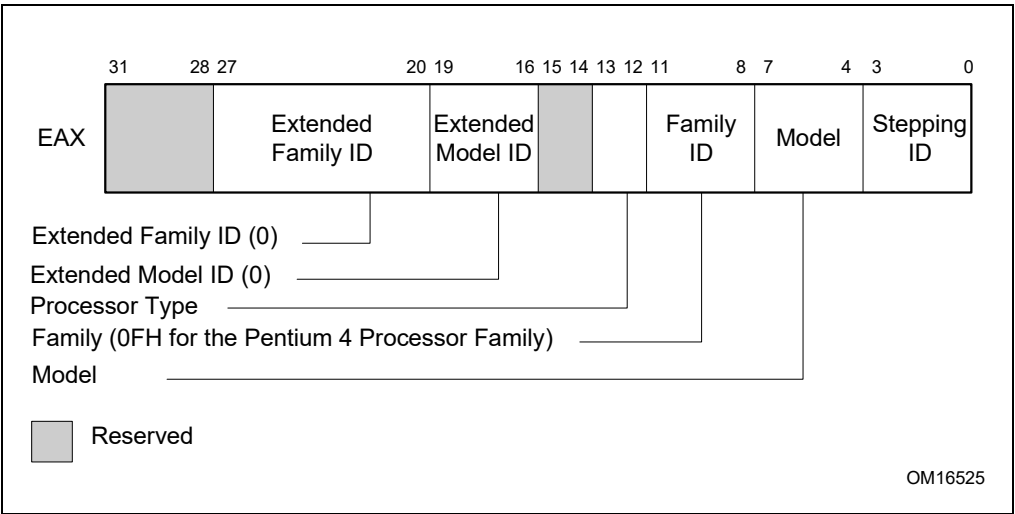
For processors that support the microcode update facility, the IA32\_BIOS\_SIGN\_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**INPUT EAX = 01H: Returns Model, Family, Stepping Information**

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-6). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

- Model — 1111B
- Family — 0101B
- Processor Type — 00B

See Table 3-9 for available processor type values. Stepping IDs are provided as needed.



**Figure 3-6. Version Information Returned by CPUID in EAX**

**Table 3-9. Processor Type Field**

Type	Encoding
Original OEM Processor	00B
Intel OverDrive™ Processor	01B
Dual processor (not applicable to Intel486 processors)	10B
Intel reserved	11B

**NOTE**

See Chapter 19 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

The Extended Family ID needs to be examined only when the Family ID is 0FH. Integrate the fields into a display using the following rule:

```
IF Family_ID ≠ 0FH
    THEN DisplayFamily = Family_ID;
    ELSE DisplayFamily = Extended_Family_ID + Family_ID;
    (* Right justify and zero-extend 4-bit field. *)
FI;
(* Show DisplayFamily as HEX field. *)
```

The Extended Model ID needs to be examined only when the Family ID is 06H or 0FH. Integrate the field into a display using the following rule:

```
IF (Family_ID = 06H or Family_ID = 0FH)
    THEN DisplayModel = (Extended_Model_ID « 4) + Model_ID;
    (* Right justify and zero-extend 4-bit field; display Model_ID as HEX field. *)
    ELSE DisplayModel = Model_ID;
FI;
(* Show DisplayModel as HEX field. *)
```

### INPUT EAX = 01H: Returns Additional Information in EBX

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

- Brand index (low byte of EBX) — this number provides an entry into a brand string table that contains brand strings for IA-32 processors. More information about this field is provided later in this section.
- CLFLUSH instruction cache line size (second byte of EBX) — this number indicates the size of the cache line flushed by the CLFLUSH and CLFLUSHOPT instructions in 8-byte increments. This field was introduced in the Pentium 4 processor.
- Local APIC ID (high byte of EBX) — this number is the 8-bit ID that is assigned to the local APIC on the processor during power up. This field was introduced in the Pentium 4 processor.

### INPUT EAX = 01H: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-10 show encodings for ECX.
- Figure 3-8 and Table 3-11 show encodings for EDX.

For all feature flags, a 1 indicates that the feature is supported. Use Intel to properly interpret feature flags.

### NOTE

Software must confirm that a processor feature is present using feature flags returned by CPUID prior to using the feature. Software should not depend on future offerings retaining all features.

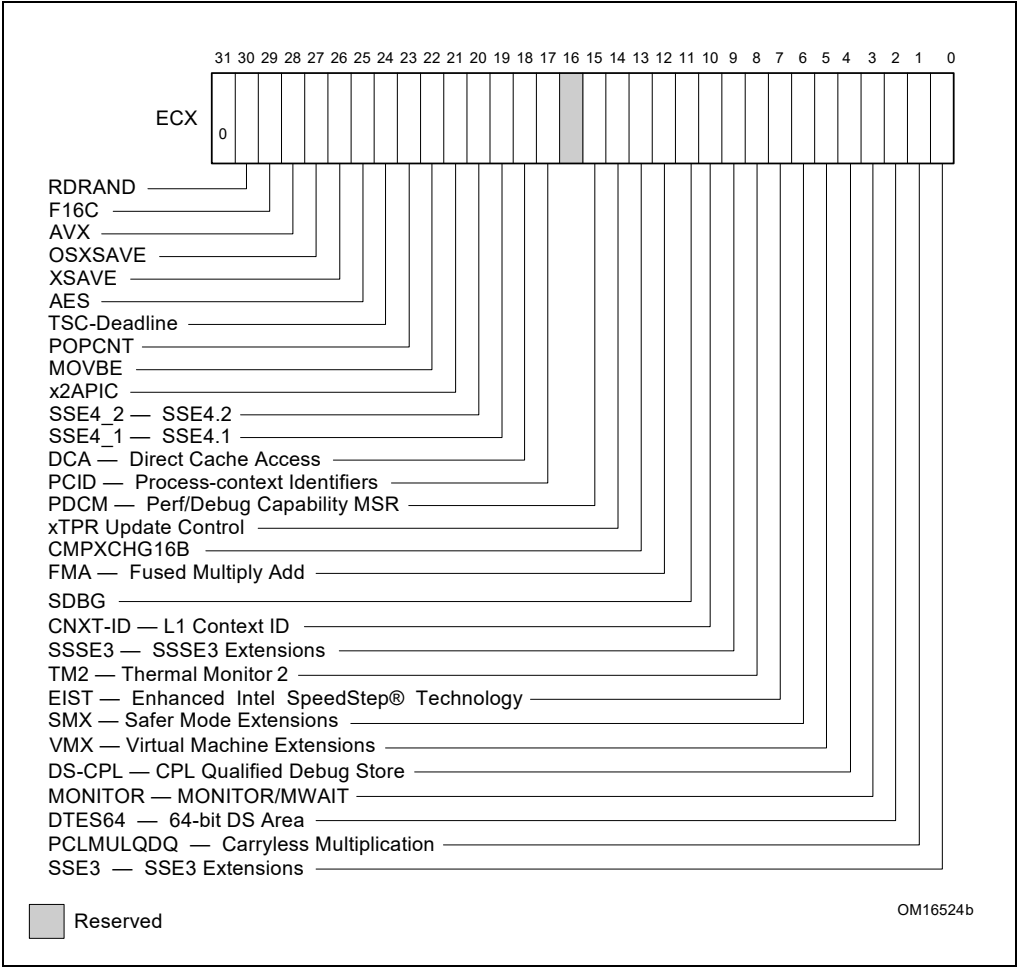


Figure 3-7. Feature Information Returned in the ECX Register

Table 3-10. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	<b>Streaming SIMD Extensions 3 (SSE3).</b> A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	<b>PCLMULQDQ.</b> A value of 1 indicates the processor supports the PCLMULQDQ instruction.
2	DTES64	<b>64-bit DS Area.</b> A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	<b>MONITOR/MWAIT.</b> A value of 1 indicates the processor supports this feature.
4	DS-CPL	<b>CPL Qualified Debug Store.</b> A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	<b>Virtual Machine Extensions.</b> A value of 1 indicates that the processor supports this technology.
6	SMX	<b>Safer Mode Extensions.</b> A value of 1 indicates that the processor supports this technology. See Chapter 6, “Safer Mode Extensions Reference”.
7	EIST	<b>Enhanced Intel SpeedStep® technology.</b> A value of 1 indicates that the processor supports this technology.
8	TM2	<b>Thermal Monitor 2.</b> A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.

Table 3-10. Feature Information Returned in the ECX Register (Contd.)

Bit #	Mnemonic	Description
10	CNXT-ID	<b>L1 Context ID.</b> A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	<b>CMPXCHG16B Available.</b> A value of 1 indicates that the feature is available. See the “CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes” section in this chapter for a description.
14	xTPR Update Control	<b>xTPR Update Control.</b> A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	<b>Perfmon and Debug Capability:</b> A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	<b>Process-context identifiers.</b> A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor’s local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always returns 0.



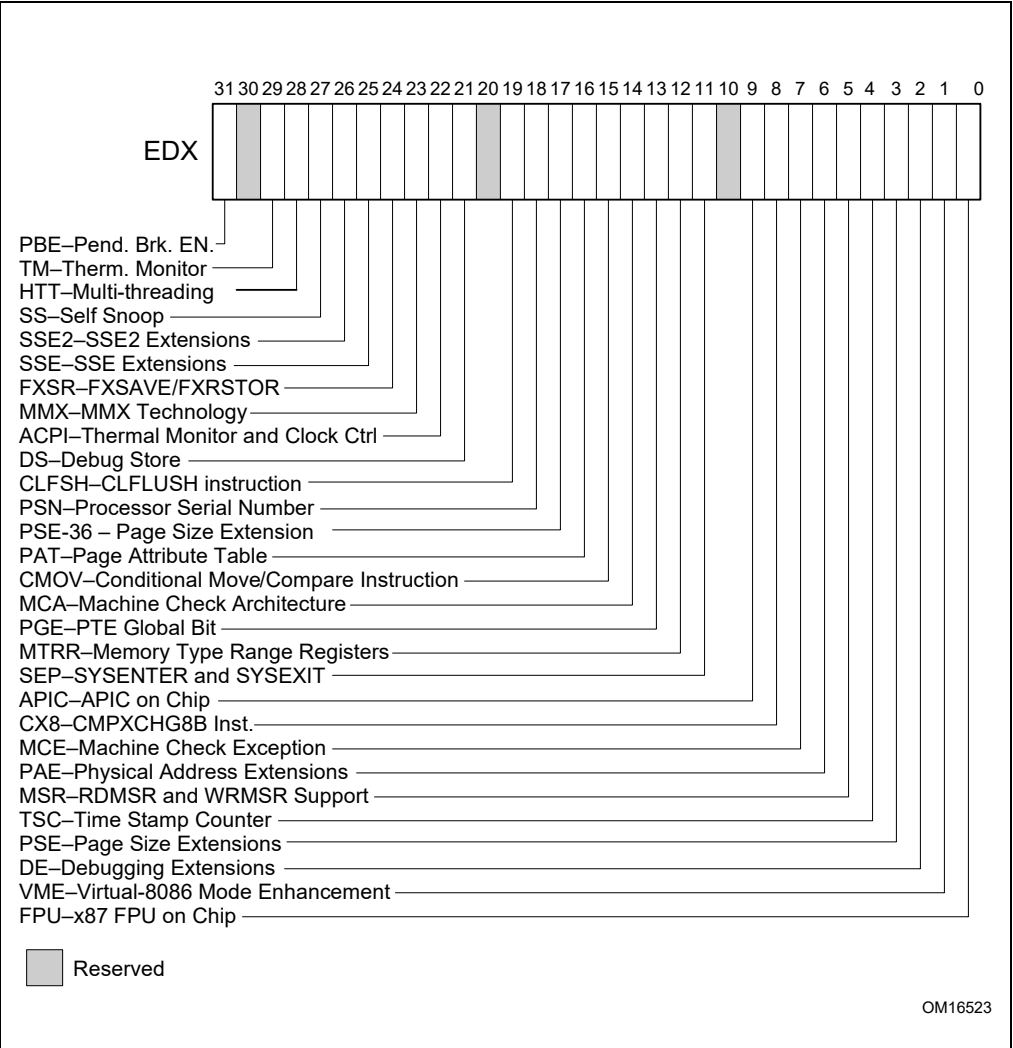


Figure 3-8. Feature Information Returned in the EDX Register

**Table 3-11. More on Feature Information Returned in the EDX Register**

Bit #	Mnemonic	Description
0	FPU	<b>Floating Point Unit On-Chip.</b> The processor contains an x87 FPU.
1	VME	<b>Virtual 8086 Mode Enhancements.</b> Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	<b>Debugging Extensions.</b> Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	<b>Page Size Extension.</b> Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	<b>Time Stamp Counter.</b> The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	<b>Model Specific Registers RDMSR and WRMSR Instructions.</b> The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	<b>Physical Address Extension.</b> Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.
7	MCE	<b>Machine Check Exception.</b> Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	<b>CMPXCHG8B Instruction.</b> The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	<b>APIC On-Chip.</b> The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	<b>SYSENTER and SYSEXIT Instructions.</b> The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	<b>Memory Type Range Registers.</b> MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	<b>Page Global Bit.</b> The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	<b>Machine Check Architecture.</b> A value of 1 indicates the Machine Check Architecture of reporting machine errors is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	<b>Conditional Move Instructions.</b> The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	<b>Page Attribute Table.</b> Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	<b>36-Bit Page Size Extension.</b> 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	<b>Processor Serial Number.</b> The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	<b>CLFLUSH Instruction.</b> CLFLUSH Instruction is supported.
20	Reserved	Reserved

**Table 3-11. More on Feature Information Returned in the EDX Register (Contd.)**

Bit #	Mnemonic	Description
21	DS	<b>Debug Store.</b> The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and processor event-based sampling (PEBS) facilities (see Chapter 23, "Introduction to Virtual-Machine Extensions," in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C</i> ).
22	ACPI	<b>Thermal Monitor and Software Controlled Clock Facilities.</b> The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	<b>Intel MMX Technology.</b> The processor supports the Intel MMX technology.
24	FXSR	<b>FXSAVE and FXRSTOR Instructions.</b> The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	<b>SSE.</b> The processor supports the SSE extensions.
26	SSE2	<b>SSE2.</b> The processor supports the SSE2 extensions.
27	SS	<b>Self Snoop.</b> The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	<b>Max APIC IDs reserved field is Valid.</b> A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	<b>Thermal Monitor.</b> The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	<b>Pending Break Enable.</b> The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

**INPUT EAX = 02H: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX**

When CPUID executes with EAX set to 02H, the processor returns information about the processor's internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-12. Table 3-12 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of "cache type" via CPUID leaf 2.

**Table 3-12. Encoding of CPUID Leaf 2 Descriptors**

Value	Type	Description
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries

**Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)**

Value	Type	Description
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLB0: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages, 128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages, 256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries
63H	TLB	Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries
64H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 512 entries
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
6AH	Cache	uTLB: 4 KByte pages, 8-way set associative, 64 entries
6BH	Cache	DTLB: 4 KByte pages, 8-way set associative, 256 entries
6CH	Cache	DTLB: 2M/4M pages, 8-way set associative, 128 entries
6DH	Cache	DTLB: 1 GByte pages, fully associative, 16 entries
70H	Cache	Trace cache: 12 K-μop, 8-way set associative
71H	Cache	Trace cache: 16 K-μop, 8-way set associative
72H	Cache	Trace cache: 32 K-μop, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size

**Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)**

Value	Type	Description
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
C0H	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries
C3H	STLB	Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GByte pages, 4-way, 16 entries.
C4H	DTLB	DTLB: 2M/4M Byte pages, 4-way associative, 32 entries
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FEH	General	CPUID leaf 2 does not report TLB descriptor information; use CPUID leaf 18H to query TLB and other address translation parameters.
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters

**Example 3-1. Example of Cache and TLB Interpretation**

The first member of the family of Pentium 4 processors returns the following information about caches and TLBs when the CPUID executes with an input value of 2:

```
EAX    66 5B 50 01H
EBX     0H
ECX     0H
EDX    00 7A 70 00H
```

Which means:

- The least-significant byte (byte 0) of register EAX is set to 01H. This value should be ignored.
- The most-significant bit of all four registers (EAX, EBX, ECX, and EDX) is set to 0, indicating that each register contains valid 1-byte descriptors.
- Bytes 1, 2, and 3 of register EAX indicate that the processor has:
  - 50H - a 64-entry instruction TLB, for mapping 4-KByte and 2-MByte or 4-MByte pages.
  - 5BH - a 64-entry data TLB, for mapping 4-KByte and 4-MByte pages.
  - 66H - an 8-KByte 1st level data cache, 4-way set associative, with a 64-Byte cache line size.
- The descriptors in registers EBX and ECX are valid, but contain NULL descriptors.
- Bytes 0, 1, 2, and 3 of register EDX indicate that the processor has:
  - 00H - NULL descriptor.
  - 70H - Trace cache: 12 K- $\mu$ op, 8-way set associative.
  - 7AH - a 256-KByte 2nd level cache, 8-way set associative, with a sector, 64-byte cache line size.
  - 00H - NULL descriptor.

**INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level**

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-8.

This Cache Size in Bytes

$$= (\text{Ways} + 1) * (\text{Partitions} + 1) * (\text{Line\_Size} + 1) * (\text{Sets} + 1)$$

$$= (\text{EBX}[31:22] + 1) * (\text{EBX}[21:12] + 1) * (\text{EBX}[11:0] + 1) * (\text{ECX} + 1)$$

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, “Multiple-Processor Management,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**INPUT EAX = 05H: Returns MONITOR and MWAIT Features**

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-8.

**INPUT EAX = 06H: Returns Thermal and Power Management Features**

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-8.



**INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information**

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-8.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-8), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

**INPUT EAX = 09H: Returns Direct Cache Access Information**

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-8.

**INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features**

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-8) is greater than Pn 0. See Table 3-8.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 23, “Introduction to Virtual-Machine Extensions,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*.

**INPUT EAX = 0BH: Returns Extended Topology Information**

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is  $\geq 0BH$ , and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-8.

**INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information**

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-8.

When CPUID executes with EAX set to 0DH and ECX = n ( $n > 1$ , and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-8. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

```
For i = 2 to 62 // sub-leaf 1 is reserved
  IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1) // VECTOR is the 64-bit value of EDX:EAX
    Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
  FI;
```

**INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Information**

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 0FH and ECX = n ( $n \geq 1$ , and is a valid ResID), the processor returns information software can use to program IA32\_PQR\_ASSOC, IA32\_QM\_EVTSEL MSRs before reading QoS data from the IA32\_QM\_CTR MSR.

**INPUT EAX = 10H: Returns Intel Resource Director Technology (Intel RDT) Allocation Enumeration Information**

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit



1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 10H and ECX = n (n >= 1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32\_resourceType\_Mask\_n.

#### **INPUT EAX = 12H: Returns Intel SGX Enumeration Information**

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-8.

#### **INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information**

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-8.

#### **INPUT EAX = 15H: Returns Time Stamp Counter and Nominal Core Crystal Clock Information**

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 3-8.

#### **INPUT EAX = 16H: Returns Processor Frequency Information**

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-8.

#### **INPUT EAX = 17H: Returns System-On-Chip Information**

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-8.

#### **INPUT EAX = 18H: Returns Deterministic Address Translation Parameters Information**

When CPUID executes with EAX set to 18H, the processor returns information about the Deterministic Address Translation Parameters. See Table 3-8.

### **METHODS FOR RETURNING BRANDING INFORMATION**

Use the following techniques to access branding information:

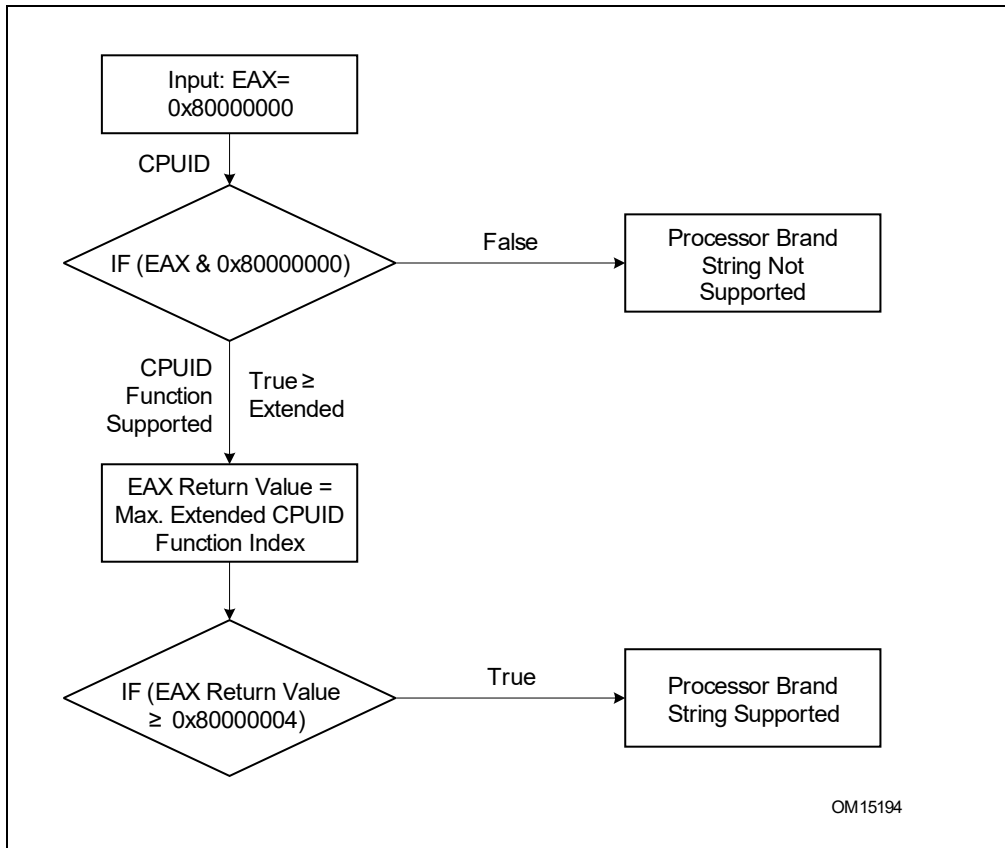
1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

#### **The Processor Brand String Method**

Figure 3-9 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

This method (introduced with Pentium 4 processors) returns an ASCII brand identification string and the Processor Base frequency of the processor to the EAX, EBX, ECX, and EDX registers.



**Figure 3-9. Determination of Support for the Processor Brand String**

### How Brand Strings Work

To use the brand string method, execute CPUID with EAX input of 80000002H through 80000004H. For each input value, CPUID returns 16 ASCII characters using EAX, EBX, ECX, and EDX. The returned string will be NULL-terminated.

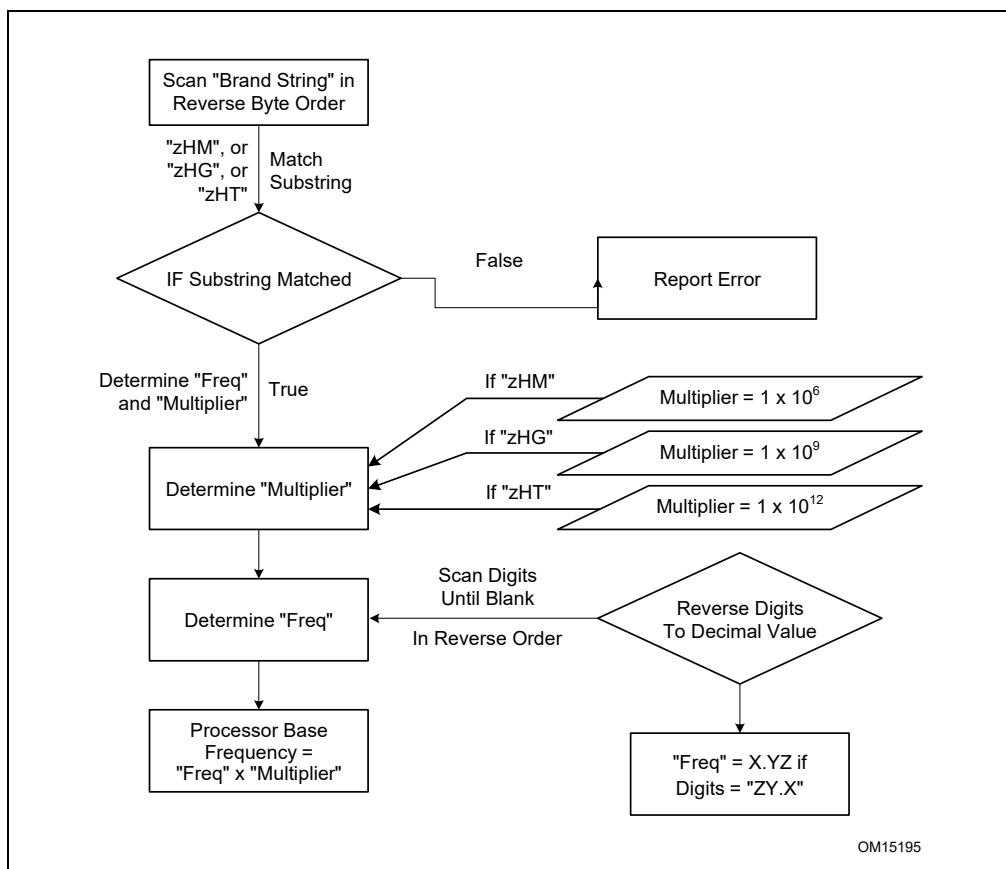
Table 3-13 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

**Table 3-13. Processor Brand String Returned with Pentium 4 Processor**

EAX Input Value	Return Values	ASCII Equivalent
80000002H	EAX = 20202020H EBX = 20202020H ECX = 20202020H EDX = 6E492020H	" " " " " " " " " "nl "
80000003H	EAX = 286C6574H EBX = 50202952H ECX = 69746E65H EDX = 52286D75H	"(let" "P )R" "itne" "R(mu"
80000004H	EAX = 20342029H EBX = 20555043H ECX = 30303531H EDX = 007A484DH	" 4 )" " UPC" "0051" "\0zHM"

### Extracting the Processor Frequency from Brand Strings

Figure 3-10 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.



**Figure 3-10. Algorithm for Extracting Processor Frequency**

## The Processor Brand Index Method

The brand index method (introduced with Pentium® III Xeon® processors) provides an entry point into a brand identification table that is maintained in memory by system software and is accessible from system- and user-level code. In this table, each brand index is associated with an ASCII brand identification string that identifies the official Intel family and model number of a processor.

When CPUID executes with EAX set to 1, the processor returns a brand index to the low byte in EBX. Software can then use this index to locate the brand identification string for the processor in the brand identification table. The first entry (brand index 0) in this table is reserved, allowing for backward compatibility with processors that do not support the brand identification feature. Starting with processor signature family ID = 0FH, model = 03H, brand index method is no longer supported. Use brand string method instead.

Table 3-14 shows brand indices that have identification strings associated with them.

**Table 3-14. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings**

Brand Index	Brand String
00H	This processor does not support the brand identification feature
01H	Intel(R) Celeron(R) processor <sup>1</sup>
02H	Intel(R) Pentium(R) III processor <sup>1</sup>
03H	Intel(R) Pentium(R) III Xeon(R) processor; If processor signature = 000006B1h, then Intel(R) Celeron(R) processor
04H	Intel(R) Pentium(R) III processor
06H	Mobile Intel(R) Pentium(R) III processor-M
07H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
08H	Intel(R) Pentium(R) 4 processor
09H	Intel(R) Pentium(R) 4 processor
0AH	Intel(R) Celeron(R) processor <sup>1</sup>
0BH	Intel(R) Xeon(R) processor; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor MP
0CH	Intel(R) Xeon(R) processor MP
0EH	Mobile Intel(R) Pentium(R) 4 processor-M; If processor signature = 00000F13h, then Intel(R) Xeon(R) processor
0FH	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
11H	Mobile Genuine Intel(R) processor
12H	Intel(R) Celeron(R) M processor
13H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
14H	Intel(R) Celeron(R) processor
15H	Mobile Genuine Intel(R) processor
16H	Intel(R) Pentium(R) M processor
17H	Mobile Intel(R) Celeron(R) processor <sup>1</sup>
18H - 0FFH	RESERVED

### NOTES:

1. Indicates versions of these processors that were introduced after the Pentium III

## IA-32 Architecture Compatibility

CPUID is not supported in early models of the Intel486 processor or in any IA-32 processor earlier than the Intel486 processor.

## Operation

IA32\_BIOS\_SIGN\_ID MSR ← Update with installed microcode revision number;

CASE (EAX) OF

EAX = 0:

EAX ← Highest basic function input value understood by CPUID;

EBX ← Vendor identification string;

EDX ← Vendor identification string;

ECX ← Vendor identification string;

BREAK;

EAX = 1H:

EAX[3:0] ← Stepping ID;

EAX[7:4] ← Model;

EAX[11:8] ← Family;

EAX[13:12] ← Processor type;

EAX[15:14] ← Reserved;

EAX[19:16] ← Extended Model;

EAX[27:20] ← Extended Family;

EAX[31:28] ← Reserved;

EBX[7:0] ← Brand Index; (\* Reserved if the value is zero. \*)

EBX[15:8] ← CLFLUSH Line Size;

EBX[16:23] ← Reserved; (\* Number of threads enabled = 2 if MT enable fuse set. \*)

EBX[24:31] ← Initial APIC ID;

ECX ← Feature flags; (\* See Figure 3-7. \*)

EDX ← Feature flags; (\* See Figure 3-8. \*)

BREAK;

EAX = 2H:

EAX ← Cache and TLB information;

EBX ← Cache and TLB information;

ECX ← Cache and TLB information;

EDX ← Cache and TLB information;

BREAK;

EAX = 3H:

EAX ← Reserved;

EBX ← Reserved;

ECX ← ProcessorSerialNumber[31:0];

(\* Pentium III processors only, otherwise reserved. \*)

EDX ← ProcessorSerialNumber[63:32];

(\* Pentium III processors only, otherwise reserved. \*)

BREAK

EAX = 4H:

EAX ← Deterministic Cache Parameters Leaf; (\* See Table 3-8. \*)

EBX ← Deterministic Cache Parameters Leaf;

ECX ← Deterministic Cache Parameters Leaf;

EDX ← Deterministic Cache Parameters Leaf;

BREAK;

EAX = 5H:

EAX ← MONITOR/MWAIT Leaf; (\* See Table 3-8. \*)

EBX ← MONITOR/MWAIT Leaf;

ECX ← MONITOR/MWAIT Leaf;

EDX ← MONITOR/MWAIT Leaf;

BREAK;

EAX = 6H:  
 EAX ← Thermal and Power Management Leaf; (\* See Table 3-8. \*)  
 EBX ← Thermal and Power Management Leaf;  
 ECX ← Thermal and Power Management Leaf;  
 EDX ← Thermal and Power Management Leaf;  
 BREAK;

EAX = 7H:  
 EAX ← Structured Extended Feature Flags Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Structured Extended Feature Flags Enumeration Leaf;  
 ECX ← Structured Extended Feature Flags Enumeration Leaf;  
 EDX ← Structured Extended Feature Flags Enumeration Leaf;  
 BREAK;

EAX = 8H:  
 EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;  
 BREAK;

EAX = 9H:  
 EAX ← Direct Cache Access Information Leaf; (\* See Table 3-8. \*)  
 EBX ← Direct Cache Access Information Leaf;  
 ECX ← Direct Cache Access Information Leaf;  
 EDX ← Direct Cache Access Information Leaf;  
 BREAK;

EAX = AH:  
 EAX ← Architectural Performance Monitoring Leaf; (\* See Table 3-8. \*)  
 EBX ← Architectural Performance Monitoring Leaf;  
 ECX ← Architectural Performance Monitoring Leaf;  
 EDX ← Architectural Performance Monitoring Leaf;  
 BREAK

EAX = BH:  
 EAX ← Extended Topology Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Extended Topology Enumeration Leaf;  
 ECX ← Extended Topology Enumeration Leaf;  
 EDX ← Extended Topology Enumeration Leaf;  
 BREAK;

EAX = CH:  
 EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;  
 BREAK;

EAX = DH:  
 EAX ← Processor Extended State Enumeration Leaf; (\* See Table 3-8. \*)  
 EBX ← Processor Extended State Enumeration Leaf;  
 ECX ← Processor Extended State Enumeration Leaf;  
 EDX ← Processor Extended State Enumeration Leaf;  
 BREAK;

EAX = EH:  
 EAX ← Reserved = 0;  
 EBX ← Reserved = 0;  
 ECX ← Reserved = 0;  
 EDX ← Reserved = 0;  
 BREAK;

EAX = FH:

EAX ← Intel Resource Director Technology Monitoring Enumeration Leaf; (\* See Table 3-8. \*)

EBX ← Intel Resource Director Technology Monitoring Enumeration Leaf;

ECX ← Intel Resource Director Technology Monitoring Enumeration Leaf;

EDX ← Intel Resource Director Technology Monitoring Enumeration Leaf;

BREAK;

EAX = 10H:

EAX ← Intel Resource Director Technology Allocation Enumeration Leaf; (\* See Table 3-8. \*)

EBX ← Intel Resource Director Technology Allocation Enumeration Leaf;

ECX ← Intel Resource Director Technology Allocation Enumeration Leaf;

EDX ← Intel Resource Director Technology Allocation Enumeration Leaf;

BREAK;

EAX = 12H:

EAX ← Intel SGX Enumeration Leaf; (\* See Table 3-8. \*)

EBX ← Intel SGX Enumeration Leaf;

ECX ← Intel SGX Enumeration Leaf;

EDX ← Intel SGX Enumeration Leaf;

BREAK;

EAX = 14H:

EAX ← Intel Processor Trace Enumeration Leaf; (\* See Table 3-8. \*)

EBX ← Intel Processor Trace Enumeration Leaf;

ECX ← Intel Processor Trace Enumeration Leaf;

EDX ← Intel Processor Trace Enumeration Leaf;

BREAK;

EAX = 15H:

EAX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf; (\* See Table 3-8. \*)

EBX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;

ECX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;

EDX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;

BREAK;

EAX = 16H:

EAX ← Processor Frequency Information Enumeration Leaf; (\* See Table 3-8. \*)

EBX ← Processor Frequency Information Enumeration Leaf;

ECX ← Processor Frequency Information Enumeration Leaf;

EDX ← Processor Frequency Information Enumeration Leaf;

BREAK;

EAX = 17H:

EAX ← System-On-Chip Vendor Attribute Enumeration Leaf; (\* See Table 3-8. \*)

EBX ← System-On-Chip Vendor Attribute Enumeration Leaf;

ECX ← System-On-Chip Vendor Attribute Enumeration Leaf;

EDX ← System-On-Chip Vendor Attribute Enumeration Leaf;

BREAK;

EAX = 18H:

EAX ← Deterministic Address Translation Parameters Enumeration Leaf; (\* See Table 3-8. \*)

EBX ← Deterministic Address Translation Parameters Enumeration Leaf;

ECX ← Deterministic Address Translation Parameters Enumeration Leaf;

EDX ← Deterministic Address Translation Parameters Enumeration Leaf;

BREAK;

EAX = 80000000H:

EAX ← Highest extended function input value understood by CPUID;

EBX ← Reserved;

ECX ← Reserved;

EDX ← Reserved;

BREAK;

EAX = 80000001H:  
     EAX ← Reserved;  
     EBX ← Reserved;  
     ECX ← Extended Feature Bits (\* See Table 3-8.\*);  
     EDX ← Extended Feature Bits (\* See Table 3-8.\*);  
 BREAK;  
 EAX = 80000002H:  
     EAX ← Processor Brand String;  
     EBX ← Processor Brand String, continued;  
     ECX ← Processor Brand String, continued;  
     EDX ← Processor Brand String, continued;  
 BREAK;  
 EAX = 80000003H:  
     EAX ← Processor Brand String, continued;  
     EBX ← Processor Brand String, continued;  
     ECX ← Processor Brand String, continued;  
     EDX ← Processor Brand String, continued;  
 BREAK;  
 EAX = 80000004H:  
     EAX ← Processor Brand String, continued;  
     EBX ← Processor Brand String, continued;  
     ECX ← Processor Brand String, continued;  
     EDX ← Processor Brand String, continued;  
 BREAK;  
 EAX = 80000005H:  
     EAX ← Reserved = 0;  
     EBX ← Reserved = 0;  
     ECX ← Reserved = 0;  
     EDX ← Reserved = 0;  
 BREAK;  
 EAX = 80000006H:  
     EAX ← Reserved = 0;  
     EBX ← Reserved = 0;  
     ECX ← Cache information;  
     EDX ← Reserved = 0;  
 BREAK;  
 EAX = 80000007H:  
     EAX ← Reserved = 0;  
     EBX ← Reserved = 0;  
     ECX ← Reserved = 0;  
     EDX ← Reserved = Misc Feature Flags;  
 BREAK;  
 EAX = 80000008H:  
     EAX ← Reserved = Physical Address Size Information;  
     EBX ← Reserved = Virtual Address Size Information;  
     ECX ← Reserved = 0;  
     EDX ← Reserved = 0;  
 BREAK;  
 EAX >= 40000000H and EAX <= 4FFFFFFFH:  
 DEFAULT: (\* EAX = Value outside of recognized range for CPUID. \*)  
     (\* If the highest basic information leaf data depend on ECX input value, ECX is honored. \*)  
     EAX ← Reserved; (\* Information returned for highest basic information leaf. \*)  
     EBX ← Reserved; (\* Information returned for highest basic information leaf. \*)  
     ECX ← Reserved; (\* Information returned for highest basic information leaf. \*)



EDX ← Reserved; (\* Information returned for highest basic information leaf. \*)

BREAK;

ESAC;

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD

If the LOCK prefix is used.

In earlier IA-32 processors that do not support the CPUID instruction, execution of the instruction results in an invalid opcode (#UD) exception being generated.

## CRC32 — Accumulate CRC32 Value

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 38 F0 /r CRC32 r32, r/m8	RM	Valid	Valid	Accumulate CRC32 on r/m8.
F2 REX 0F 38 F0 /r CRC32 r32, r/m8*	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 0F 38 F1 /r CRC32 r32, r/m16	RM	Valid	Valid	Accumulate CRC32 on r/m16.
F2 0F 38 F1 /r CRC32 r32, r/m32	RM	Valid	Valid	Accumulate CRC32 on r/m32.
F2 REX.W 0F 38 F0 /r CRC32 r64, r/m8	RM	Valid	N.E.	Accumulate CRC32 on r/m8.
F2 REX.W 0F 38 F1 /r CRC32 r64, r/m64	RM	Valid	N.E.	Accumulate CRC32 on r/m64.

### NOTES:

\*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

### Description

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 11EDC6F41H) value for the second operand (source operand) and stores the result in the destination operand. The source operand can be a register or a memory location. The destination operand must be an r32 or r64 register. If the destination is an r64 register, then the 32-bit result is stored in the least significant double word and 00000000H is stored in the most significant double word of the r64 register.

The initial value supplied in the destination operand is a double word integer stored in the r32 register or the least significant double word of the r64 register. To incrementally accumulate a CRC32 value, software retains the result of the previous CRC32 operation in the destination operand, then executes the CRC32 instruction again with new input data in the source operand. Data contained in the source operand is processed in reflected bit order. This means that the most significant bit of the source operand is treated as the least significant bit of the quotient, and so on, for all the bits of the source operand. Likewise, the result of the CRC operation is stored in the destination operand in reflected bit order. This means that the most significant bit of the resulting CRC (bit 31) is stored in the least significant bit of the destination operand (bit 0), and so on, for all the bits of the CRC.

### Operation

#### Notes:

BIT\_REFLECT64: DST[63-0] = SRC[0-63]  
 BIT\_REFLECT32: DST[31-0] = SRC[0-31]  
 BIT\_REFLECT16: DST[15-0] = SRC[0-15]  
 BIT\_REFLECT8: DST[7-0] = SRC[0-7]  
 MOD2: Remainder from Polynomial division modulus 2

CRC32 instruction for 64-bit source operand and 64-bit destination operand:

```

TEMP1[63-0] ← BIT_REFLECT64 (SRC[63-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[95-0] ← TEMP1[63-0] « 32
TEMP4[95-0] ← TEMP2[31-0] « 64
TEMP5[95-0] ← TEMP3[95-0] XOR TEMP4[95-0]
TEMP6[31-0] ← TEMP5[95-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
DEST[63-32] ← 00000000H

```

CRC32 instruction for 32-bit source operand and 32-bit destination operand:

```

TEMP1[31-0] ← BIT_REFLECT32 (SRC[31-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[63-0] ← TEMP1[31-0] « 32
TEMP4[63-0] ← TEMP2[31-0] « 32
TEMP5[63-0] ← TEMP3[63-0] XOR TEMP4[63-0]
TEMP6[31-0] ← TEMP5[63-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

```

CRC32 instruction for 16-bit source operand and 32-bit destination operand:

```

TEMP1[15-0] ← BIT_REFLECT16 (SRC[15-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[47-0] ← TEMP1[15-0] « 32
TEMP4[47-0] ← TEMP2[31-0] « 16
TEMP5[47-0] ← TEMP3[47-0] XOR TEMP4[47-0]
TEMP6[31-0] ← TEMP5[47-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

```

CRC32 instruction for 8-bit source operand and 64-bit destination operand:

```

TEMP1[7-0] ← BIT_REFLECT8 (SRC[7-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] ← TEMP1[7-0] « 32
TEMP4[39-0] ← TEMP2[31-0] « 8
TEMP5[39-0] ← TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] ← TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
DEST[63-32] ← 00000000H

```

CRC32 instruction for 8-bit source operand and 32-bit destination operand:

```

TEMP1[7-0] ← BIT_REFLECT8 (SRC[7-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] ← TEMP1[7-0] « 32
TEMP4[39-0] ← TEMP2[31-0] « 8
TEMP5[39-0] ← TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] ← TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

```

### Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

unsigned int \_mm\_crc32\_u8( unsigned int crc, unsigned char data )  
 unsigned int \_mm\_crc32\_u16( unsigned int crc, unsigned short data )  
 unsigned int \_mm\_crc32\_u32( unsigned int crc, unsigned int data )  
 unsigned \_\_int64 \_mm\_crc32\_u64( unsigned \_\_int64 crc, unsigned \_\_int64 data )

### SIMD Floating Point Exceptions

None

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.  
 #SS(0) If a memory operand effective address is outside the SS segment limit.  
 #PF (fault-code) For a page fault.  
 #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.  
 #UD If CPUID.01H:ECX.SSE4\_2 [Bit 20] = 0.  
 If LOCK prefix is used.

### Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.  
 #SS(0) If a memory operand effective address is outside the SS segment limit.  
 #UD If CPUID.01H:ECX.SSE4\_2 [Bit 20] = 0.  
 If LOCK prefix is used.

### Virtual 8086 Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.  
 #SS(0) If a memory operand effective address is outside the SS segment limit.  
 #PF (fault-code) For a page fault.  
 #AC(0) If alignment checking is enabled and an unaligned memory reference is made.  
 #UD If CPUID.01H:ECX.SSE4\_2 [Bit 20] = 0.  
 If LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

### 64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.  
 #SS(0) If a memory address referencing the SS segment is in a non-canonical form.  
 #PF (fault-code) For a page fault.  
 #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.  
 #UD If CPUID.01H:ECX.SSE4\_2 [Bit 20] = 0.  
 If LOCK prefix is used.

## CVTDP2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F E6 /r CVTDP2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1.
EVEX.128.F3.0F.W0 E6 /r VCVTDQ2PD xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert 2 packed signed doubleword integers from xmm2/m128/m32bcst to eight packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W0 E6 /r VCVTDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W0 E6 /r VCVTDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two, four or eight packed signed doubleword integers in the source operand (the second operand) to two, four or eight packed double-precision floating-point values in the destination operand (the first operand).

EVEX encoded versions: The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

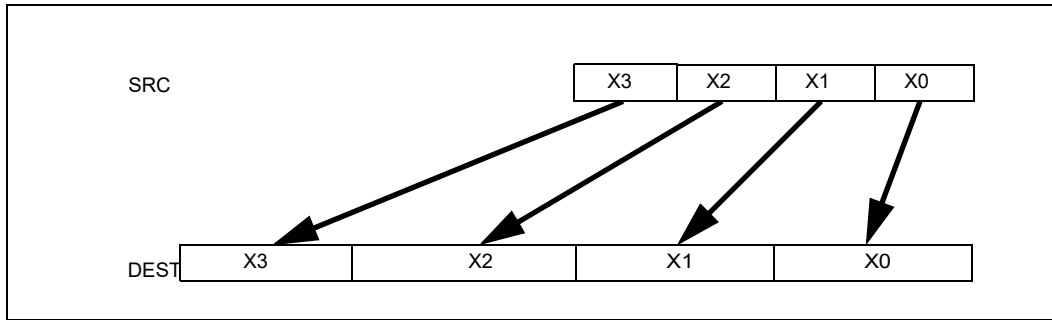


Figure 3-11. CVTDQ2PD (VEX.256 encoded version)

### Operation

**VCVTDQ2PD (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTDQ2PD (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    FI;
  ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VCVTDQ2PD (VEX.256 encoded version)**

```

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:256] ← 0

```

**VCVTDQ2PD (VEX.128 encoded version)**

```

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] ← 0

```

**CVTDQ2PD (128-bit Legacy SSE version)**

```

DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTDQ2PD __m512d __mm512_cvtepi32_pd( __m256i a);
VCVTDQ2PD __m512d __mm512_mask_cvtepi32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m512d __mm512_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m256d __mm256_cvtepi32_pd( __m128i src);
VCVTDQ2PD __m256d __mm256_mask_cvtepi32_pd( __m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m256d __mm256_maskz_cvtepi32_pd( __mmask8 k, __m256i a);
VCVTDQ2PD __m128d __mm_mask_cvtepi32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD __m128d __mm_maskz_cvtepi32_pd( __mmask8 k, __m128i a);
CVTDQ2PD __m128d __mm_cvtepi32_pd( __m128i src)

```

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 5;

EVEX-encoded instructions, see Exceptions Type E5.

#UD                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.



## CVTDDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5B /r CVTDDQ2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.
VEX.128.0F.WIG 5B /r VCVTDDQ2PS xmm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.
VEX.256.0F.WIG 5B /r VCVTDDQ2PS ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single-precision floating-point values in ymm1.
EVEX.128.0F.W0 5B /r VCVTDDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W0 5B /r VCVTDDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single-precision floating-point values in ymm1 with writemask k1.
EVEX.512.0F.W0 5B /r VCVTDDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single-precision floating-point values in the destination operand.

EVEX encoded versions: The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

**Operation****VCVTDQ2PS (EVEX encoded versions) when SRC operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC); ; refer to Table 2-4 in the *Intel® Architecture Instruction Set Extensions Programming Reference*

ELSE

SET\_RM(MXCSR.RM); ; refer to Table 2-4 in the *Intel® Architecture Instruction Set Extensions Programming Reference*

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTDQ2PS (EVEX encoded versions) when SRC operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])

ELSE

DEST[i+31:i] ←

Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTDQ2PS (VEX.256 encoded version)**

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[63:32] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[95:64] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[127:96] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[127:96])  
 DEST[159:128] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[159:128])  
 DEST[191:160] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[191:160])  
 DEST[223:192] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[223:192])  
 DEST[255:224] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[255:224])  
 DEST[MAXVL-1:256] ← 0

**VCVTDQ2PS (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[63:32] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[95:64] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[127:96] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[127:96])  
 DEST[MAXVL-1:128] ← 0

**CVTDQ2PS (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0])  
 DEST[63:32] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:32])  
 DEST[95:64] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[95:64])  
 DEST[127:96] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[127:96])  
 DEST[MAXVL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTDQ2PS \_\_m512 \_\_mm512\_cvtepi32\_ps(\_\_m512i a);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_mask\_cvtepi32\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512i a);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_maskz\_cvtepi32\_ps(\_\_mmask16 k, \_\_m512i a);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_cvt\_roundepsi32\_ps(\_\_m512i a, int r);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_mask\_cvt\_roundepsi32\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512i a, int r);  
 VCVTDQ2PS \_\_m512 \_\_mm512\_maskz\_cvt\_roundepsi32\_ps(\_\_mmask16 k, \_\_m512i a, int r);  
 VCVTDQ2PS \_\_m256 \_\_mm256\_mask\_cvtepi32\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256i a);  
 VCVTDQ2PS \_\_m256 \_\_mm256\_maskz\_cvtepi32\_ps(\_\_mmask8 k, \_\_m256i a);  
 VCVTDQ2PS \_\_m128 \_\_mm\_mask\_cvtepi32\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128i a);  
 VCVTDQ2PS \_\_m128 \_\_mm\_maskz\_cvtepi32\_ps(\_\_mmask8 k, \_\_m128i a);  
 CVTDQ2PS \_\_m256 \_\_mm256\_cvtepi32\_ps(\_\_m256i src)  
 CVTDQ2PS \_\_m128 \_\_mm\_cvtepi32\_ps(\_\_m128i src)

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.
EVEX.128.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.
EVEX.512.F2.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

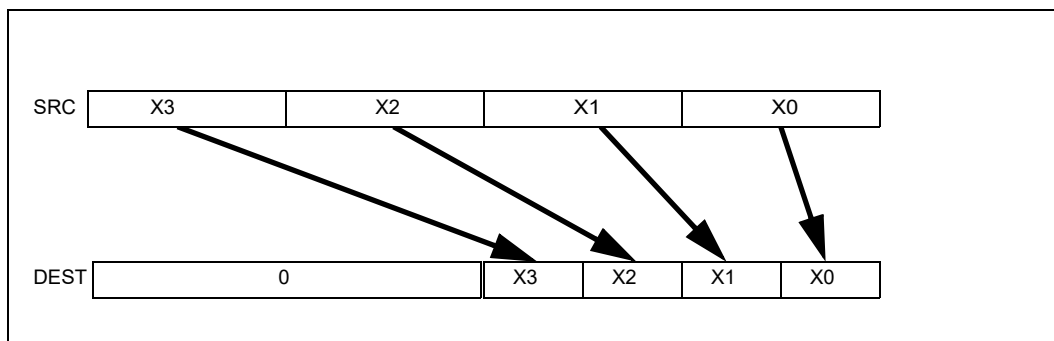


Figure 3-12. VCVTPD2DQ (VEX.256 encoded version)

### Operation

**VCVTPD2DQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0

**VCVTPD2DQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
                    Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
                ELSE
                    DEST[i+31:i] ←
                    Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0

```

**VCVTPD2DQ (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192])
DEST[MAXVL-1:128] ← 0

```

**VCVTPD2DQ (VEX.128 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[MAXVL-1:64] ← 0

```

**CVTPD2DQ (128-bit Legacy SSE version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] ← 0
DEST[MAXVL-1:128] (unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPD2DQ __m256i _mm512_cvtpd_epi32( __m512d a);
VCVTPD2DQ __m256i _mm512_mask_cvtpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_maskz_cvtpd_epi32( __mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_cvt_roundpd_epi32( __m512d a, int r);
VCVTPD2DQ __m256i _mm512_mask_cvt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m256i _mm512_maskz_cvt_roundpd_epi32( __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m128i _mm256_mask_cvtpd_epi32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm256_maskz_cvtpd_epi32( __mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm_mask_cvtpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm_maskz_cvtpd_epi32( __mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm256_cvtpd_epi32( __m256d src)
CVTPD2DQ __m128i _mm_cvtpd_epi32( __m128d src)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

See Exceptions Type 2; additionally

EVEX-encoded instructions, see Exceptions Type E2.

#UD                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTPD2PI—Convert Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2D /r CVTPD2PI <i>mm, xmm/m128</i>	RM	Valid	Valid	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[127:64]);
```

### Intel C/C++ Compiler Intrinsic Equivalent

CVTPD1PI: `__m64 _mm_cvtpd_pi32(__m128d a)`

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

See Table 22-4, “Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.



## CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.
VEX.128.66.0F.WIG 5A /r VCVTPD2PS xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.
VEX.256.66.0F.WIG 5A /r VCVTPD2PS xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four single-precision floating-point values in xmm1.
EVEX.128.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.66.0F.W1 5A /r VCVTPD2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight single-precision floating-point values in ymm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two, four or eight packed double-precision floating-point values in the source operand (second operand) to two, four or eight packed single-precision floating-point values in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64-bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

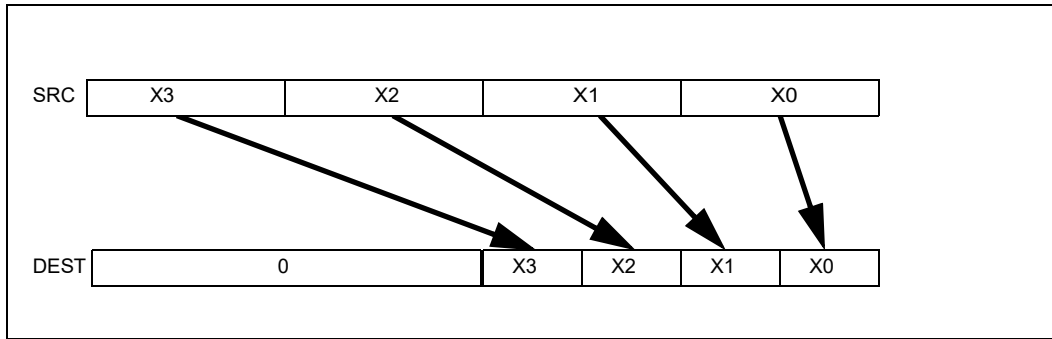


Figure 3-13. VCVTPD2PS (VEX.256 encoded version)

### Operation

**VCVTPD2PS (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

DEST[i+31:i] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Single\_Precision\_Floating\_Point(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0

**VCVTPD2PS (EVEX encoded version) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_Single_Precision_Floating_Point(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ELSE
      FI;
  ENDFOR
DEST[MAXVL-1:VL/2] ← 0

```

**VCVTPD2PS (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192])
DEST[MAXVL-1:128] ← 0

```

**VCVTPD2PS (VEX.128 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAXVL-1:64] ← 0

```

**CVTPD2PS (128-bit Legacy SSE version)**

```

DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] ← 0
DEST[MAXVL-1:128] (unmodified)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTPD2PS __m256 _mm512_cvtpd_ps( __m512d a);
VCVTPD2PS __m256 _mm512_mask_cvtpd_ps( __m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_maskz_cvtpd_ps( __mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_cvt_roundpd_ps( __m512d a, int r);
VCVTPD2PS __m256 _mm512_mask_cvt_roundpd_ps( __m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 _mm512_maskz_cvt_roundpd_ps( __mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 _mm256_mask_cvtpd_ps( __m128 s, __mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm256_maskz_cvtpd_ps( __mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm_mask_cvtpd_ps( __m128 s, __mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm_maskz_cvtpd_ps( __mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm256_cvtpd_ps( __m256d a)
CVTPD2PS __m128 _mm_cvtpd_ps( __m128d a)

```

### SIMD Floating-Point Exceptions

Invalid, Precision, Underflow, Overflow, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTPI2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2A /r CVTPI2PD <i>xmm, mm/m64*</i>	RM	Valid	Valid	Convert two packed signed doubleword integers from <i>mm/mem64</i> to two packed double-precision floating-point values in <i>xmm</i> .

### NOTES:

\*Operation is different for different operand sets; see the Description section.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed double-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. In addition, depending on the operand configuration:

- For operands *xmm, mm*: the instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PD instruction is executed.
- For operands *xmm, m64*: the instruction does not cause a transition to MMX technology and does not take x87 FPU exceptions.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[31:0]);

DEST[127:64] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[63:32]);

### Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PD: `__m128d _mm_cvtpi32_pd(__m64 a)`

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Table 22-6, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

## CVTPI2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF 2A /r CVTPI2PS <i>xmm, mm/m64</i>	RM	Valid	Valid	Convert two signed doubleword integers from <i>mm/m64</i> to two single-precision floating-point values in <i>xmm</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two packed signed doubleword integers in the source operand (second operand) to two packed single-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
(* High quadword of destination unchanged *)
```

### Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PS: `__m128 _mm_cvtpi32_ps(__m128 a, __m64 b)`

### SIMD Floating-Point Exceptions

Precision

### Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

## CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.
EVEX.128.66.0F.W0 5B /r VCVTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 5B /r VCVTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 5B /r VCVTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

## Operation

### VCVTPS2DQ (encoded versions) when src operand is a register

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

### VCVTPS2DQ (EVEX encoded versions) when src operand is a memory source

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO 15

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])

ELSE

DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0



**VCVTPS2DQ (VEX.256 encoded version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[159:128] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[159:128])  
 DEST[191:160] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[191:160])  
 DEST[223:192] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[223:192])  
 DEST[255:224] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[255:224])

**VCVTPS2DQ (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[MAXVL-1:128] ← 0

**CVTPS2DQ (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[127:96])  
 DEST[MAXVL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPS2DQ \_\_m512i \_\_mm512\_cvtps\_epi32( \_\_m512 a);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_mask\_cvtps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvtps\_epi32( \_\_mmask16 k, \_\_m512 a);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_cvt\_roundps\_epi32( \_\_m512 a, int r);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_mask\_cvt\_roundps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a, int r);  
 VCVTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvt\_roundps\_epi32( \_\_mmask16 k, \_\_m512 a, int r);  
 VCVTPS2DQ \_\_m256i \_\_mm256\_mask\_cvtps\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256 a);  
 VCVTPS2DQ \_\_m256i \_\_mm256\_maskz\_cvtps\_epi32( \_\_mmask8 k, \_\_m256 a);  
 VCVTPS2DQ \_\_m128i \_\_mm\_mask\_cvtps\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128 a);  
 VCVTPS2DQ \_\_m128i \_\_mm\_maskz\_cvtps\_epi32( \_\_mmask8 k, \_\_m128 a);  
 VCVTPS2DQ \_\_m256i \_\_mm256\_cvtps\_epi32( \_\_m256 a)  
 CVTPS2DQ \_\_m128i \_\_mm\_cvtps\_epi32( \_\_m128 a)

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5A /r CVTTPS2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.
VEX.128.0F.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.
VEX.256.0F.WIG 5A /r VCVTPS2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed single-precision floating-point values in xmm2/m128 to four packed double-precision floating-point values in ymm1.
EVEX.128.0F.W0 5A /r VCVTPS2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	AVX512VL AVX512F	Convert two packed single-precision floating-point values in xmm2/m64/m32bcst to packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W0 5A /r VCVTPS2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL	Convert four packed single-precision floating-point values in xmm2/m128/m32bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.0F.W0 5A /r VCVTPS2PD zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	B	V/V	AVX512F	Convert eight packed single-precision floating-point values in ymm2/m256/b32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two, four or eight packed single-precision floating-point values in the source operand (second operand) to two, four or eight packed double-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 64-bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64-bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

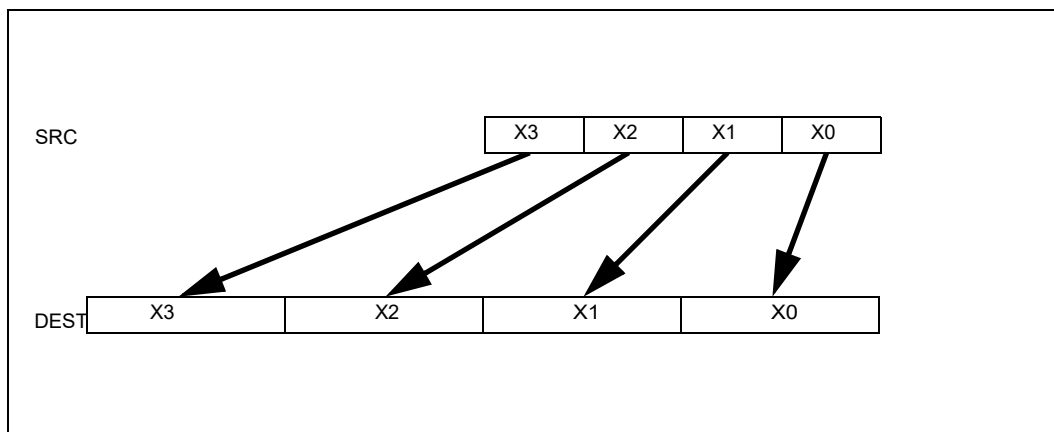


Figure 3-14. CVTSP2PD (VEX.256 encoded version)

### Operation

**VCVTPS2PD (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTPS2PD (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1)

        THEN

          DEST[i+63:i] ←

          Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0])

        ELSE

          DEST[i+63:i] ←

          Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

      FI;

    ELSE

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VCVTPS2PD (VEX.256 encoded version)**

```

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96])
DEST[MAXVL-1:256] ← 0

```

**VCVTPS2PD (VEX.128 encoded version)**

```

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] ← 0

```

**CVTPS2PD (128-bit Legacy SSE version)**

```

DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPS2PD __m512d __mm512_cvtps_pd( __m256 a);
VCVTPS2PD __m512d __mm512_mask_cvtps_pd( __m512d s, __mmask8 k, __m256 a);
VCVTPS2PD __m512d __mm512_maskz_cvtps_pd( __mmask8 k, __m256 a);
VCVTPS2PD __m512d __mm512_cvt_roundps_pd( __m256 a, int sae);
VCVTPS2PD __m512d __mm512_mask_cvt_roundps_pd( __m512d s, __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m512d __mm512_maskz_cvt_roundps_pd( __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m256d __mm256_mask_cvtps_pd( __m256d s, __mmask8 k, __m128 a);
VCVTPS2PD __m256d __mm256_maskz_cvtps_pd( __mmask8 k, __m128a);
VCVTPS2PD __m128d __mm_mask_cvtps_pd( __m128d s, __mmask8 k, __m128 a);
VCVTPS2PD __m128d __mm_maskz_cvtps_pd( __mmask8 k, __m128 a);
VCVTPS2PD __m256d __mm256_cvtps_pd( __m128 a)
CVTPS2PD __m128d __mm_cvtps_pd( __m128 a)

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTPS2PI—Convert Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF 2D /r CVTPS2PI <i>mm, xmm/m64</i>	RM	Valid	Valid	Convert two packed single-precision floating-point values from <i>xmm/m64</i> to two packed signed doubleword integers in <i>mm</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand).

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

CVTPS2PI causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);
```

### Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PI: `__m64 _mm_cvtps_pi32(__m128 a)`

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

## CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2D /r CVTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
F2 REX.W 0F 2D /r CVTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
VEX.LIG.F2.0F.W0 2D /r <sup>1</sup> VCVTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
VEX.LIG.F2.0F.W1 2D /r <sup>1</sup> VCVTSD2SI r64, xmm1/m64	A	V/N.E. <sup>2</sup>	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
EVEX.LIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64{er}	B	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
EVEX.LIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64{er}	B	V/N.E. <sup>2</sup>	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.

### NOTES:

1. Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000\_00000000H) is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VCVTSD2SI (EVEX encoded version)**

IF SRC \*is register\* AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode and OperandSize = 64

THEN DEST[63:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0]);

ELSE DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0]);

FI

**(V)CVTSD2SI**

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer(SRC[63:0]);

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSD2SI int \_mm\_cvtsd\_i32(\_\_m128d);

VCVTSD2SI int \_mm\_cvt\_roundsd\_i32(\_\_m128d, int r);

VCVTSD2SI \_\_int64 \_mm\_cvtsd\_i64(\_\_m128d);

VCVTSD2SI \_\_int64 \_mm\_cvt\_roundsd\_i64(\_\_m128d, int r);

CVTSD2SI \_\_int64 \_mm\_cvtsd\_si64(\_\_m128d);

CVTSD2SI int \_mm\_cvtsd\_si32(\_\_m128d a)

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	A	V/V	SSE2	Convert one double-precision floating-point value in xmm2/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1, xmm2, xmm3/m64	B	V/V	AVX	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2.
EVEX.NDS.LIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a double-precision floating-point value in the “convert-from” source operand (the second operand in SSE2 version, otherwise the third operand) to a single-precision floating-point value in the destination operand.

When the “convert-from” operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result is written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.



**Operation****VCVTSD2SS (EVEX encoded version)**

IF (SRC2 \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAXVL-1:128] ← 0

**VCVTSD2SS (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC2[63:0]);

DEST[127:32] ← SRC1[127:32]

DEST[MAXVL-1:128] ← 0

**CVTSD2SS (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Double\_Precision\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

(\* DEST[MAXVL-1:32] Unmodified \*)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSD2SS \_\_m128 \_\_mm\_mask\_cvtssd\_ss(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128d b);

VCVTSD2SS \_\_m128 \_\_mm\_maskz\_cvtssd\_ss(\_\_mmask8 k, \_\_m128 a, \_\_m128d b);

VCVTSD2SS \_\_m128 \_\_mm\_cvt\_roundsd\_ss(\_\_m128 a, \_\_m128d b, int r);

VCVTSD2SS \_\_m128 \_\_mm\_mask\_cvt\_roundsd\_ss(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128d b, int r);

VCVTSD2SS \_\_m128 \_\_mm\_maskz\_cvt\_roundsd\_ss(\_\_mmask8 k, \_\_m128 a, \_\_m128d b, int r);

CVTSD2SS \_\_m128 \_\_mm\_cvtssd\_ss(\_\_m128 a, \_\_m128d b)

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2A /r CVTSI2SD xmm1, r32/m32	A	V/V	SSE2	Convert one signed doubleword integer from r32/m32 to one double-precision floating-point value in xmm1.
F2 REX.W 0F 2A /r CVTSI2SD xmm1, r/m64	A	V/N.E.	SSE2	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64	B	V/N.E. <sup>1</sup>	AVX	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	C	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64{er}	C	V/N.E. <sup>1</sup>	AVX512F	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

VEX.W1 and EVEX.W1 versions: promotes the instruction to use 64-bit input value in 64-bit mode.

Software should ensure VCVTSI2SD is encoded with VEX.L=0. Encoding VCVTSI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VCVTSI2SD (EVEX encoded version)**

IF (SRC2 \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAXVL-1:128] ← 0

**VCVTSI2SD (VEX.128 encoded version)**

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAXVL-1:128] ← 0

**CVTSI2SD**

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[63:0] ← Convert\_Integer\_To\_Double\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSI2SD \_\_m128d\_mm\_cvti32\_sd(\_\_m128d s, int a);

VCVTSI2SD \_\_m128d\_mm\_cvti64\_sd(\_\_m128d s, \_\_int64 a);

VCVTSI2SD \_\_m128d\_mm\_cvt\_roundi64\_sd(\_\_m128d s, \_\_int64 a, int r);

CVTSI2SD \_\_m128d\_mm\_cvtsi64\_sd(\_\_m128d s, \_\_int64 a);

CVTSI2SD \_\_m128d\_mm\_cvtsi32\_sd(\_\_m128d a, int b)

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3 if W1, else Type 5.

EVEX-encoded instructions, see Exceptions Type E3NF if W1, else Type E10NF.

## CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2A /r CVTSI2SS xmm1, r/m32	A	V/V	SSE	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
F3 REX.W 0F 2A /r CVTSI2SS xmm1, r/m64	A	V/N.E.	SSE	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64	B	V/N.E. <sup>1</sup>	AVX	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32{er}	C	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64{er}	C	V/N.E. <sup>1</sup>	AVX512F	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.

### NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the “convert-from” source operand to a single-precision floating-point value in the destination operand (first operand). The “convert-from” source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The “convert-from” source operand (the second operand) is a general-purpose register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: the converted result is written to the low doubleword element of the destination under the writemask.

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VCVTSI2SS (EVEX encoded version)**

IF (SRC2 \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAXVL-1:128] ← 0

**VCVTSI2SS (VEX.128 encoded version)**

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAXVL-1:128] ← 0

**CVTSI2SS (128-bit Legacy SSE version)**

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Integer\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[MAXVL-1:32] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSI2SS \_\_m128\_mm\_cvti32\_ss(\_\_m128 s, int a);

VCVTSI2SS \_\_m128\_mm\_cvt\_roundi32\_ss(\_\_m128 s, int a, int r);

VCVTSI2SS \_\_m128\_mm\_cvti64\_ss(\_\_m128 s, \_\_int64 a);

VCVTSI2SS \_\_m128\_mm\_cvt\_roundi64\_ss(\_\_m128 s, \_\_int64 a, int r);

CVTSI2SS \_\_m128\_mm\_cvtsi64\_ss(\_\_m128 s, \_\_int64 a);

CVTSI2SS \_\_m128\_mm\_cvtsi32\_ss(\_\_m128 a, int b);

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3NF.

## CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	A	V/V	SSE2	Convert one single-precision floating-point value in xmm2/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	B	V/V	AVX	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2.
EVEX.NDS.LIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a single-precision floating-point value in the “convert-from” source operand to a double-precision floating-point value in the destination operand. When the “convert-from” source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The “convert-from” source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The “convert-from” source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### VCVTSS2SD (EVEX encoded version)

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] = 0

FI;

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAXVL-1:128] ← 0

**VCVTSS2SD (VEX.128 encoded version)**

DEST[63:0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0])

DEST[127:64] ← SRC1[127:64]

DEST[MAXVL-1:128] ← 0

**CVTSS2SD (128-bit Legacy SSE version)**

DEST[63:0] ← Convert\_Single\_Precision\_To\_Double\_Precision\_Floating\_Point(SRC[31:0]);

DEST[MAXVL-1:64] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2SD \_\_m128d \_mm\_cvt\_roundss\_sd(\_\_m128d a, \_\_m128 b, int r);

VCVTSS2SD \_\_m128d \_mm\_mask\_cvt\_roundss\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128 b, int r);

VCVTSS2SD \_\_m128d \_mm\_maskz\_cvt\_roundss\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128 a, int r);

VCVTSS2SD \_\_m128d \_mm\_mask\_cvtss\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128 b);

VCVTSS2SD \_\_m128d \_mm\_maskz\_cvtss\_sd(\_\_mmask8 m, \_\_m128d a, \_\_m128 b);

CVTSS2SD \_\_m128d \_mm\_cvtss\_sd(\_\_m128d a, \_\_m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2D /r CVTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
F3 REX.W 0F 2D /r CVTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
VEX.LIG.F3.0F.W0 2D /r <sup>1</sup> VCVTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
VEX.LIG.F3.0F.W1 2D /r <sup>1</sup> VCVTSS2SI r64, xmm1/m32	A	V/N.E. <sup>2</sup>	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
EVEX.LIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32{er}	B	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
EVEX.LIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32{er}	B	V/N.E. <sup>2</sup>	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.

### NOTES:

- Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
- VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where w represents the number of bits in the destination format) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.



**Operation****VCVTSS2SI (EVEX encoded version)**

IF (SRC \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

FI;

**(V)CVTSS2SI (Legacy and VEX.128 encoded version)**

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer(SRC[31:0]);

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2SI int \_\_mm\_cvtss\_i32( \_\_m128 a);

VCVTSS2SI int \_\_mm\_cvt\_roundss\_i32( \_\_m128 a, int r);

VCVTSS2SI \_\_int64 \_\_mm\_cvtss\_i64( \_\_m128 a);

VCVTSS2SI \_\_int64 \_\_mm\_cvt\_roundss\_i64( \_\_m128 a, int r);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.128.66.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.256.66.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.
EVEX.128.66.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}	B	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two, four or eight packed double-precision floating-point values in the source operand (second operand) to two, four or eight packed signed doubleword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:64) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

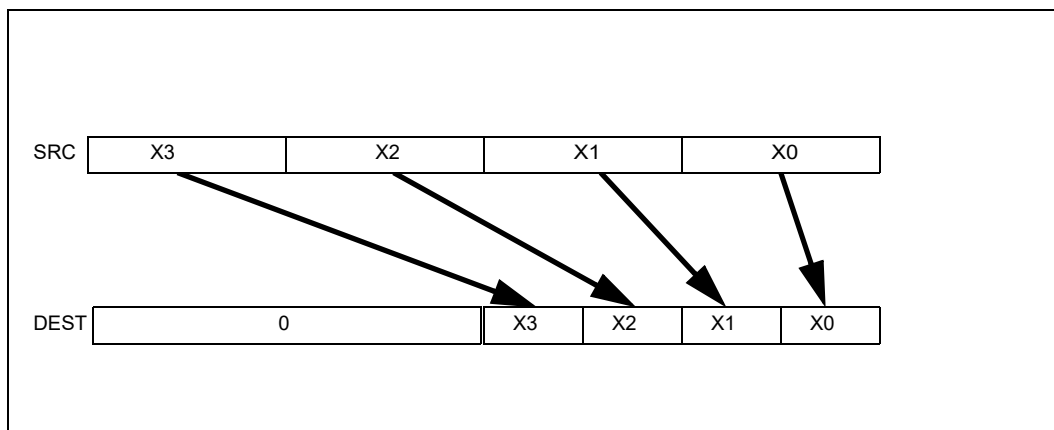


Figure 3-15. VCVTTPD2DQ (VEX.256 encoded version)

**Operation**

**VCVTTPD2DQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0

```

**VCVTTPD2DQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF;
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0

```

**VCVTTPD2DQ (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192])
DEST[MAXVL-1:128] ← 0

```

**VCVTTPD2DQ (VEX.128 encoded version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[MAXVL-1:64] ← 0

```

**CVTTPD2DQ (128-bit Legacy SSE version)**

```

DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] ← 0
DEST[MAXVL-1:128] (unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPD2DQ __m256i __mm512_cvttpd_epi32( __m512d a);
VCVTTPD2DQ __m256i __mm512_mask_cvttpd_epi32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i __mm512_maskz_cvttpd_epi32( __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i __mm512_cvtt_roundpd_epi32( __m512d a, int sae);
VCVTTPD2DQ __m256i __mm512_mask_cvtt_roundpd_epi32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m256i __mm512_maskz_cvtt_roundpd_epi32( __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m128i __mm256_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i __mm256_maskz_cvttpd_epi32( __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i __mm_mask_cvttpd_epi32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i __mm_maskz_cvttpd_epi32( __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i __mm256_cvttpd_epi32( __m256d src);
CVTTPD2DQ __m128i __mm_cvttpd_epi32( __m128d src);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTTPD2PI—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2C /r CVTTPD2PI <i>mm, xmm/m128</i>	RM	Valid	Valid	Convert two packed double-precision floating-point values from <i>xmm/m128</i> to two packed signed doubleword integers in <i>mm</i> using truncation.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPD2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[63:0]);
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32_
               Truncate(SRC[127:64]);
```

### Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD1PI: `__m64 __mm_cvttpd_pi32(__m128d a)`

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Mode Exceptions

See Table 22-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

## CVTTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5B /r CVTTTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.
EVEX.128.F3.0F.W0 5B /r VCVTTTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.0F.W0 5B /r VCVTTTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.0F.W0 5B /r VCVTTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	B	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

**Operation****VCVTTTPS2DQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VCVTTTPS2DQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO 15
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VCVTTTPS2DQ (VEX.256 encoded version)**

```

DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[159:128] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[159:128])
DEST[191:160] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[191:160])
DEST[223:192] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[223:192])
DEST[255:224] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[255:224])

```



**VCVTTPS2DQ (VEX.128 encoded version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[127:96])  
 DEST[MAXVL-1:128] ← 0

**CVTTPS2DQ (128-bit Legacy SSE version)**

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0])  
 DEST[63:32] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:32])  
 DEST[95:64] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[95:64])  
 DEST[127:96] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[127:96])  
 DEST[MAXVL-1:128] (unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTPS2DQ \_\_m512i \_\_mm512\_cvttps\_epi32( \_\_m512 a);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_mask\_cvttps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvttps\_epi32( \_\_mmask16 k, \_\_m512 a);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_cvtt\_roundps\_epi32( \_\_m512 a, int sae);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_mask\_cvtt\_roundps\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512 a, int sae);  
 VCVTTPS2DQ \_\_m512i \_\_mm512\_maskz\_cvtt\_roundps\_epi32( \_\_mmask16 k, \_\_m512 a, int sae);  
 VCVTTPS2DQ \_\_m256i \_\_mm256\_mask\_cvttps\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256 a);  
 VCVTTPS2DQ \_\_m256i \_\_mm256\_maskz\_cvttps\_epi32( \_\_mmask8 k, \_\_m256 a);  
 VCVTTPS2DQ \_\_m128i \_\_mm128\_mask\_cvttps\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128 a);  
 VCVTTPS2DQ \_\_m128i \_\_mm128\_maskz\_cvttps\_epi32( \_\_mmask8 k, \_\_m128 a);  
 VCVTTPS2DQ \_\_m256i \_\_mm256\_cvttps\_epi32( \_\_m256 a)  
 CVTTPS2DQ \_\_m128i \_\_mm128\_cvttps\_epi32( \_\_m128 a)

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2; additionally

EVEX-encoded instructions, see Exceptions Type E2.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## CVTTPS2PI—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF 2C /r CVTTPS2PI <i>mm, xmm/m64</i>	RM	Valid	Valid	Convert two single-precision floating-point values from <i>xmm/m64</i> to two signed doubleword signed integers in <i>mm</i> using truncation.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);
```

### Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI: `__m64 _mm_cvttps_pi32(__m128 a)`

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

See Table 22-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

## CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2C /r CVTTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
VEX.LIG.F2.0F.W0 2C /r <sup>1</sup> VCVTTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F2.0F.W1 2C /r <sup>1</sup> VCVTTSD2SI r64, xmm1/m64	B	V/N.E. <sup>2</sup>	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64{sae}	B	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64{sae}	B	V/N.E. <sup>2</sup>	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.

### NOTES:

1. Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000\_00000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

### (V)CVTTSD2SI (All versions)

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[63:0]);

FI;

## Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2SI int \_\_mm\_cvttssd\_i32( \_\_m128d a);

VCVTTSD2SI int \_\_mm\_cvtt\_roundssd\_i32( \_\_m128d a, int sae);

VCVTTSD2SI \_\_int64 \_\_mm\_cvttssd\_i64( \_\_m128d a);

VCVTTSD2SI \_\_int64 \_\_mm\_cvtt\_roundssd\_i64( \_\_m128d a, int sae);

CVTTSD2SI int \_\_mm\_cvttssd\_si32( \_\_m128d a);

CVTTSD2SI \_\_int64 \_\_mm\_cvttssd\_si64( \_\_m128d a);

## SIMD Floating-Point Exceptions

Invalid, Precision

## Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
VEX.LIG.F3.0F.W0 2C /r <sup>1</sup> VCVTTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F3.0F.W1 2C /r <sup>1</sup> VCVTTSS2SI r64, xmm1/m32	A	V/N.E. <sup>2</sup>	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32{sae}	B	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32{sae}	B	V/N.E. <sup>2</sup>	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.

### NOTES:

- Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
- For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H or 80000000\_00000000H if operand size is 64 bits) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****(V)CVTTSS2SI (All versions)**

IF 64-Bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_Integer\_Truncate(SRC[31:0]);

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTSS2SI int \_\_mm\_cvtss\_i32( \_\_m128 a);

VCVTTSS2SI int \_\_mm\_cvtt\_roundss\_i32( \_\_m128 a, int sae);

VCVTTSS2SI \_\_int64 \_\_mm\_cvtss\_i64( \_\_m128 a);

VCVTTSS2SI \_\_int64 \_\_mm\_cvtt\_roundss\_i64( \_\_m128 a, int sae);

CVTTSS2SI int \_\_mm\_cvtss\_si32( \_\_m128 a);

CVTTSS2SI \_\_int64 \_\_mm\_cvtss\_si64( \_\_m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

See Exceptions Type 3; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

**CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
99	CWD	Z0	Valid	Valid	DX:AX ← sign-extend of AX.
99	CDQ	Z0	Valid	Valid	EDX:EAX ← sign-extend of EAX.
REX.W + 99	CQO	Z0	Valid	N.E.	RDX:RAX ← sign-extend of RAX.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Doubles the size of the operand in register AX, EAX, or RAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX, EDX:EAX, or RDX:RAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register. The CQO instruction (available in 64-bit mode only) copies the sign (bit 63) of the value in the RAX register into every bit position in the RDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before word division. The CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CQO instruction can be used to produce a double quadword dividend from a quadword before a quadword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. The CQO mnemonics reference the same opcode as CWD/CDQ. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

```
IF OperandSize = 16 (* CWD instruction *)
  THEN
    DX ← SignExtend(AX);
  ELSE IF OperandSize = 32 (* CDQ instruction *)
    EDX ← SignExtend(EAX); FI;
  ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
    RDX ← SignExtend(RAX); FI;
FI;
```

**Flags Affected**

None

**Exceptions (All Operating Modes)**

#UD If the LOCK prefix is used.

## DAA—Decimal Adjust AL after Addition

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
27	DAA	ZO	Invalid	Valid	Decimal adjust AL after addition.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

### Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

IF 64-Bit Mode

THEN

#UD;

ELSE

old\_AL ← AL;

old\_CF ← CF;

CF ← 0;

IF (((AL AND 0FH) > 9) or AF = 1)

THEN

AL ← AL + 6;

CF ← old\_CF or (Carry from AL ← AL + 6);

AF ← 1;

ELSE

AF ← 0;

FI;

IF ((old\_AL > 99H) or (old\_CF = 1))

THEN

AL ← AL + 60H;

CF ← 1;

ELSE

CF ← 0;

FI;

FI;

### Example

```

ADD  AL, BL  Before: AL=79H BL=35H EFLAGS(OSZAPC)=XXXXXX
                After: AL=AEH BL=35H EFLAGS(OSZAPC)=110000
DAA                                Before: AL=AEH BL=35H EFLAGS(OSZAPC)=110000
                After: AL=14H BL=35H EFLAGS(OSZAPC)=X00111
DAA                                Before: AL=2EH BL=35H EFLAGS(OSZAPC)=110000
                After: AL=34H BL=35H EFLAGS(OSZAPC)=X00101

```



### Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see the “Operation” section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#UD If the LOCK prefix is used.

### Compatibility Mode Exceptions

#UD If the LOCK prefix is used.

### 64-Bit Mode Exceptions

#UD If in 64-bit mode.

## DAS—Decimal Adjust AL after Subtraction

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
2F	DAS	Z0	Invalid	Valid	Decimal adjust AL after subtraction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

### Operation

```

IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        old_AL ← AL;
        old_CF ← CF;
        CF ← 0;
        IF ((AL AND 0FH) > 9) or AF = 1
            THEN
                AL ← AL - 6;
                CF ← old_CF or (Borrow from AL ← AL - 6);
                AF ← 1;
            ELSE
                AF ← 0;
        FI;
        IF ((old_AL > 99H) or (old_CF = 1))
            THEN
                AL ← AL - 60H;
                CF ← 1;
        FI;
    FI;

```

### Example

```

SUB    AL, BL    Before: AL = 35H, BL = 47H, EFLAGS(OSZAPC) = XXXXXX
                  After: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
DAA                      Before: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
                  After: AL = 88H, BL = 47H, EFLAGS(OSZAPC) = X10111

```

### Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal borrow in either digit of the result (see the “Operation” section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

### Protected Mode Exceptions

#UD                      If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD                      If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#UD                      If the LOCK prefix is used.

### Compatibility Mode Exceptions

#UD                      If the LOCK prefix is used.

### 64-Bit Mode Exceptions

#UD                      If in 64-bit mode.

## DEC—Decrement by 1

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FE /1	DEC <i>r/m8</i>	M	Valid	Valid	Decrement <i>r/m8</i> by 1.
REX + FE /1	DEC <i>r/m8</i>	M	Valid	N.E.	Decrement <i>r/m8</i> by 1.
FF /1	DEC <i>r/m16</i>	M	Valid	Valid	Decrement <i>r/m16</i> by 1.
FF /1	DEC <i>r/m32</i>	M	Valid	Valid	Decrement <i>r/m32</i> by 1.
REX.W + FF /1	DEC <i>r/m64</i>	M	Valid	N.E.	Decrement <i>r/m64</i> by 1.
48+rw	DEC <i>r16</i>	O	N.E.	Valid	Decrement <i>r16</i> by 1.
48+rd	DEC <i>r32</i>	O	N.E.	Valid	Decrement <i>r32</i> by 1.

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m ( <i>r, w</i> )	NA	NA	NA
O	opcode + rd ( <i>r, w</i> )	NA	NA	NA

### Description

Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, DEC *r16* and DEC *r32* are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← DEST - 1;

### Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

### Protected Mode Exceptions

#GP(0)	If the destination operand is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## DIV—Unsigned Divide

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /6	DIV <i>r/m8</i>	M	Valid	Valid	Unsigned divide AX by <i>r/m8</i> , with result stored in AL ← Quotient, AH ← Remainder.
REX + F6 /6	DIV <i>r/m8</i> *	M	Valid	N.E.	Unsigned divide AX by <i>r/m8</i> , with result stored in AL ← Quotient, AH ← Remainder.
F7 /6	DIV <i>r/m16</i>	M	Valid	Valid	Unsigned divide DX:AX by <i>r/m16</i> , with result stored in AX ← Quotient, DX ← Remainder.
F7 /6	DIV <i>r/m32</i>	M	Valid	Valid	Unsigned divide EDX:EAX by <i>r/m32</i> , with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /6	DIV <i>r/m64</i>	M	Valid	N.E.	Unsigned divide RDX:RAX by <i>r/m64</i> , with result stored in RAX ← Quotient, RDX ← Remainder.

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-15.

**Table 3-15. DIV Action**

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	<i>r/m8</i>	AL	AH	255
Doubleword/word	DX:AX	<i>r/m16</i>	AX	DX	65,535
Quadword/doubleword	EDX:EAX	<i>r/m32</i>	EAX	EDX	$2^{32} - 1$
Doublequadword/quadword	RDX:RAX	<i>r/m64</i>	RAX	RDX	$2^{64} - 1$

**Operation**

```

IF SRC = 0
    THEN #DE; FI; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
    THEN
        temp ← AX / SRC;
        IF temp > FFH
            THEN #DE; (* Divide error *)
            ELSE
                AL ← temp;
                AH ← AX MOD SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp ← DX:AX / SRC;
            IF temp > FFFFH
                THEN #DE; (* Divide error *)
            ELSE
                AX ← temp;
                DX ← DX:AX MOD SRC;
            FI;
        FI;
    ELSE IF Operandsize = 32 (* Quadword/doubleword operation *)
        THEN
            temp ← EDX:EAX / SRC;
            IF temp > FFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                EAX ← temp;
                EDX ← EDX:EAX MOD SRC;
            FI;
        FI;
    ELSE IF 64-Bit Mode and Operandsize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp ← RDX:RAX / SRC;
            IF temp > FFFFFFFFFFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                RAX ← temp;
                RDX ← RDX:RAX MOD SRC;
            FI;
        FI;
    FI;
FI;

```

**Flags Affected**

The CF, OF, SF, ZF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

#DE	If the source operand (divisor) is 0 If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#DE	If the source operand (divisor) is 0. If the quotient is too large for the designated register.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#DE	If the source operand (divisor) is 0. If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#DE	If the source operand (divisor) is 0 If the quotient is too large for the designated register.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.



## DIVPD—Divide Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5E /r DIVPD xmm1, xmm2/m128	A	V/V	SSE2	Divide packed double-precision floating-point values in xmm1 by packed double-precision floating-point values in xmm2/mem.
VEX.NDS.128.66.0F.WIG 5E /r VDIVPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/mem.
VEX.NDS.256.66.0F.WIG 5E /r VDIVPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem.
EVEX.NDS.128.66.0F.W1 5E /r VDIVPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 5E /r VDIVPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 5E /r VDIVPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Divide packed double-precision floating-point values in zmm2 by packed double-precision FP values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD divide of the double-precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand (the second operand) is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

VEX.128 encoded version: The first source operand (the second operand) is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination are zeroed.

128-bit Legacy SSE version: The second source operand (the second operand) can be an XMM register or an 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAXVL-1:128) of the corresponding destination are unmodified.

## Operation

### VDIVPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC); ; refer to Table 2-4 in the *Intel® Architecture Instruction Set Extensions Programming Reference*

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← SRC1[i+63:i] / SRC2[63:0]

ELSE

DEST[i+63:i] ← SRC1[i+63:i] / SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

### VDIVPD (VEX.256 encoded version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[191:128] ← SRC1[191:128] / SRC2[191:128]

DEST[255:192] ← SRC1[255:192] / SRC2[255:192]

DEST[MAXVL-1:256] ← 0;

### VDIVPD (VEX.128 encoded version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[MAXVL-1:128] ← 0;

### DIVPD (128-bit Legacy SSE version)

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]

DEST[127:64] ← SRC1[127:64] / SRC2[127:64]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VDIVPD __m512d _mm512_div_pd( __m512d a, __m512d b);
VDIVPD __m512d _mm512_mask_div_pd( __m512d s, __mmask8 k, __m512d a, __m512d b);
VDIVPD __m512d _mm512_maskz_div_pd( __mmask8 k, __m512d a, __m512d b);
VDIVPD __m256d _mm256_mask_div_pd( __m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd( __mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd( __m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd( __mmask8 k, __m128d a, __m128d b);
VDIVPD __m512d _mm512_div_round_pd( __m512d a, __m512d b, int);
VDIVPD __m512d _mm512_mask_div_round_pd( __m512d s, __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m512d _mm512_maskz_div_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m256d _mm256_div_pd( __m256d a, __m256d b);
DIVPD __m128d _mm_div_pd( __m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## DIVPS—Divide Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5E /r DIVPS xmm1, xmm2/m128	A	V/V	SSE	Divide packed single-precision floating-point values in xmm1 by packed single-precision floating-point values in xmm2/mem.
VEX.NDS.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/mem.
VEX.NDS.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/mem.
EVEX.NDS.128.0F.W0 5E /r VDIVPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 5E /r VDIVPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F	Divide packed single-precision floating-point values in zmm2 by packed single-precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD divide of the four, eight or sixteen packed single-precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single-precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation****VDIVPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC1[i+31:i] / SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC1[i+31:i] / SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VDIVPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]

DEST[63:32] ← SRC1[63:32] / SRC2[63:32]

DEST[95:64] ← SRC1[95:64] / SRC2[95:64]

DEST[127:96] ← SRC1[127:96] / SRC2[127:96]

DEST[159:128] ← SRC1[159:128] / SRC2[159:128]

DEST[191:160] ← SRC1[191:160] / SRC2[191:160]

DEST[223:192] ← SRC1[223:192] / SRC2[223:192]

DEST[255:224] ← SRC1[255:224] / SRC2[255:224].

DEST[MAXVL-1:256] ← 0;

**VDIVPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]

DEST[63:32] ← SRC1[63:32] / SRC2[63:32]

DEST[95:64] ← SRC1[95:64] / SRC2[95:64]

DEST[127:96] ← SRC1[127:96] / SRC2[127:96]

DEST[MAXVL-1:128] ← 0

**DIVPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]  
 DEST[63:32] ← SRC1[63:32] / SRC2[63:32]  
 DEST[95:64] ← SRC1[95:64] / SRC2[95:64]  
 DEST[127:96] ← SRC1[127:96] / SRC2[127:96]  
 DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VDIVPS \_\_m512 \_\_mm512\_div\_ps( \_\_m512 a, \_\_m512 b);  
 VDIVPS \_\_m512 \_\_mm512\_mask\_div\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VDIVPS \_\_m512 \_\_mm512\_maskz\_div\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VDIVPD \_\_m256d \_\_mm256\_mask\_div\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VDIVPD \_\_m256d \_\_mm256\_maskz\_div\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256d b);  
 VDIVPD \_\_m128d \_\_mm\_mask\_div\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VDIVPD \_\_m128d \_\_mm\_maskz\_div\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b);  
 VDIVPS \_\_m512 \_\_mm512\_div\_round\_ps( \_\_m512 a, \_\_m512 b, int);  
 VDIVPS \_\_m512 \_\_mm512\_mask\_div\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VDIVPS \_\_m512 \_\_mm512\_maskz\_div\_round\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b, int);  
 VDIVPS \_\_m256 \_\_mm256\_div\_ps (\_\_m256 a, \_\_m256 b);  
 DIVPS \_\_m128 \_\_mm\_div\_ps (\_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## DIVSD—Divide Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD xmm1, xmm2/m64	A	V/V	SSE2	Divide low double-precision floating-point value in xmm1 by low double-precision floating-point value in xmm2/m64.
VEX.NDS.LIG.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.
EVEX.NDS.LIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VDIVSD is encoded with VEX.L=0. Encoding VDIVSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

### VDIVSD (EVEX encoded version)

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

### VDIVSD (VEX.128 encoded version)

```

DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

### DIVSD (128-bit Legacy SSE version)

```

DEST[63:0] ← DEST[63:0] / SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd(__mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd(__m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd(__mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd(__m128d a, __m128d b);

```

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

## Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.



## DIVSS—Divide Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS xmm1, xmm2/m32	A	V/V	SSE	Divide low single-precision floating-point value in xmm1 by low single-precision floating-point value in xmm2/m32.
VEX.NDS.LIG.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.
EVEX.NDS.LIG.F3.0F.W0 5E /r VDIVSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VDIVSS is encoded with VEX.L=0. Encoding VDIVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

### VDIVSS (EVEX encoded version)

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

### VDIVSS (VEX.128 encoded version)

```

DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

### DIVSS (128-bit Legacy SSE version)

```

DEST[31:0] ← DEST[31:0] / SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

VDIVSS __m128 _mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_maskz_div_ss(__mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_div_round_ss(__m128 a, __m128 b, int);
VDIVSS __m128 _mm_mask_div_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 _mm_maskz_div_round_ss(__mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 _mm_div_ss(__m128 a, __m128 b);

```

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

## Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## DPPD — Dot Product of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 41 /r ib DPPD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Selectively multiply packed DP floating-point values from <i>xmm1</i> with packed DP floating-point values from <i>xmm2</i> , add and selectively store the packed DP floating-point values to <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 41 /r ib VDPPD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Selectively multiply packed DP floating-point values from <i>xmm2</i> with packed DP floating-point values from <i>xmm3</i> , add and selectively store the packed DP floating-point values to <i>xmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Conditionally multiplies the packed double-precision floating-point values in the destination operand (first operand) with the packed double-precision floating-point values in the source (second operand) depending on a mask extracted from bits [5:4] of the immediate operand (third operand). If a condition mask bit is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The two resulting double-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [1:0] of the immediate byte.

If a broadcast mask bit is “1”, the intermediate result is copied to the corresponding qword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPD follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

If VDPPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

## Operation

### DP\_primitive (SRC1, SRC2)

```
IF (imm8[4] = 1)
    THEN Temp1[63:0] ← DEST[63:0] * SRC[63:0]; // update SIMD exception flags
    ELSE Temp1[63:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[127:64] ← DEST[127:64] * SRC[127:64]; // update SIMD exception flags
    ELSE Temp1[127:64] ← +0.0; FI;
/* if unmasked exception reported, execute exception handler*/
```

```
Temp2[63:0] ← Temp1[63:0] + Temp1[127:64]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
```

```
IF (imm8[0] = 1)
    THEN DEST[63:0] ← Temp2[63:0];
    ELSE DEST[63:0] ← +0.0; FI;
IF (imm8[1] = 1)
    THEN DEST[127:64] ← Temp2[63:0];
    ELSE DEST[127:64] ← +0.0; FI;
```

### DPPD (128-bit Legacy SSE version)

```
DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] (Unmodified)
```

### VDPPD (VEX.128 encoded version)

```
DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] ← 0
```

## Flags Affected

None

## Intel C/C++ Compiler Intrinsic Equivalent

```
DPPD:   __m128d _mm_dp_pd ( __m128d a, __m128d b, const int mask);
```

## SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation. Unmasked exceptions will leave the destination untouched.

## Other Exceptions

See Exceptions Type 2; additionally

#UD If VEX.L = 1.

## DPPS — Dot Product of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 40 /r ib DPPS <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Selectively multiply packed SP floating-point values from <i>xmm1</i> with packed SP floating-point values from <i>xmm2</i> , add and selectively store the packed SP floating-point values or zero values to <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 40 /r ib VDPPS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Multiply packed SP floating point values from <i>xmm1</i> with packed SP floating point values from <i>xmm2/mem</i> selectively add and store to <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 40 /r ib VDPPS <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX	Multiply packed single-precision floating-point values from <i>ymm2</i> with packed SP floating point values from <i>ymm3/mem</i> , selectively add pairs of elements and store to <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Conditionally multiplies the packed single precision floating-point values in the destination operand (first operand) with the packed single-precision floats in the source (second operand) depending on a mask extracted from the high 4 bits of the immediate byte (third operand). If a condition mask bit in *Imm8*[7:4] is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The four resulting single-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [3:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding dword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

DPPS follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

## Operation

### DP\_primitive (SRC1, SRC2)

```

IF (imm8[4] = 1)
    THEN Temp1[31:0] ← DEST[31:0] * SRC[31:0]; // update SIMD exception flags
    ELSE Temp1[31:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[63:32] ← DEST[63:32] * SRC[63:32]; // update SIMD exception flags
    ELSE Temp1[63:32] ← +0.0; FI;
IF (imm8[6] = 1)
    THEN Temp1[95:64] ← DEST[95:64] * SRC[95:64]; // update SIMD exception flags
    ELSE Temp1[95:64] ← +0.0; FI;
IF (imm8[7] = 1)
    THEN Temp1[127:96] ← DEST[127:96] * SRC[127:96]; // update SIMD exception flags
    ELSE Temp1[127:96] ← +0.0; FI;

Temp2[31:0] ← Temp1[31:0] + Temp1[63:32]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp3[31:0] ← Temp1[95:64] + Temp1[127:96]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp4[31:0] ← Temp2[31:0] + Temp3[31:0]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

```

```

IF (imm8[0] = 1)
    THEN DEST[31:0] ← Temp4[31:0];
    ELSE DEST[31:0] ← +0.0; FI;
IF (imm8[1] = 1)
    THEN DEST[63:32] ← Temp4[31:0];
    ELSE DEST[63:32] ← +0.0; FI;
IF (imm8[2] = 1)
    THEN DEST[95:64] ← Temp4[31:0];
    ELSE DEST[95:64] ← +0.0; FI;
IF (imm8[3] = 1)
    THEN DEST[127:96] ← Temp4[31:0];
    ELSE DEST[127:96] ← +0.0; FI;

```

### DPPS (128-bit Legacy SSE version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] (Unmodified)

```

### VDPPS (VEX.128 encoded version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[MAXVL-1:128] ← 0

```

### VDPPS (VEX.256 encoded version)

```

DEST[127:0] ← DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128] ← DP_Primitive(SRC1[255:128], SRC2[255:128]);

```

## Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

(V)DPPS: `__m128 _mm_dp_ps ( __m128 a, __m128 b, const int mask);`

VDPPS: `__m256 _mm256_dp_ps ( __m256 a, __m256 b, const int mask);`

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation, in the order of their execution. Unmasked exceptions will leave the destination operands unchanged.

### Other Exceptions

See Exceptions Type 2.

## EMMS—Empty MMX Technology State

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF 77	EMMS	Z0	Valid	Valid	Set the x87 FPU tag word to empty.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Sets the values of all the tags in the x87 FPU tag word to empty (all 1s). This operation marks the x87 FPU data registers (which are aliased to the MMX technology registers) as available for use by x87 FPU floating-point instructions. (See Figure 8-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the format of the x87 FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in x87 FPU tag word to valid (all 0s).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

EMMS operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$x87FPUTagWord \leftarrow FFFFH;$

### Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_empty()
```

### Flags Affected

None

### Protected Mode Exceptions

#UD If CR0.EM[bit 2] = 1.  
 #NM If CR0.TS[bit 3] = 1.  
 #MF If there is a pending FPU exception.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.



**ENTER—Make Stack Frame for Procedure Parameters**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C8 <i>iw</i> 00	ENTER <i>imm16</i> , 0	II	Valid	Valid	Create a stack frame for a procedure.
C8 <i>iw</i> 01	ENTER <i>imm16</i> , 1	II	Valid	Valid	Create a stack frame with a nested pointer for a procedure.
C8 <i>iw</i> ib	ENTER <i>imm16</i> , <i>imm8</i>	II	Valid	Valid	Create a stack frame with nested pointers for a procedure.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
II	<i>iw</i>	<i>imm8</i>	NA	NA

**Description**

Creates a stack frame (comprising of space for dynamic storage and 1-32 frame pointer storage) for a procedure. The first operand (*imm16*) specifies the size of the dynamic storage in the stack frame (that is, the number of bytes of dynamically allocated on the stack for the procedure). The second operand (*imm8*) gives the lexical nesting level (0 to 31) of the procedure. The nesting level (*imm8* mod 32) and the *OperandSize* attribute determine the size in bytes of the storage space for frame pointers.

The nesting level determines the number of frame pointers that are copied into the “display area” of the new stack frame from the preceding frame. The default size of the frame pointer is the *StackAddrSize* attribute, but can be overridden using the 66H prefix. Thus, the *OperandSize* attribute determines the size of each frame pointer that will be copied into the stack frame and the data being transferred from SP/ESP/RSP register into the BP/EBP/RBP register.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See “Procedure Calls for Block-Structured Languages” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information about the actions of the ENTER instruction.

The ENTER instruction causes a page fault whenever a write using the final value of the stack pointer (within the current stack segment) would do so.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded. Use of 66H prefix changes frame pointer operand size to 16 bits.

When the 66H prefix is used and causing the *OperandSize* attribute to be less than the *StackAddrSize*, software is responsible for the following:

- The companion LEAVE instruction must also use the 66H prefix,
- The value in the RBP/EBP register prior to executing “66H ENTER” must be within the same 16KByte region of the current stack pointer (RSP/ESP), such that the value of RBP/EBP after “66H ENTER” remains a valid address in the stack. This ensures “66H LEAVE” can restore 16-bits of data from the stack.

**Operation**

```

AllocSize ← imm16;
NestingLevel ← imm8 MOD 32;
IF (OperandSize = 64)
    THEN
        Push(RBP); (* RSP decrements by 8 *)
        FrameTemp ← RSP;
    ELSE IF OperandSize = 32
        THEN
            Push(EBP); (* (E)SP decrements by 4 *)
            FrameTemp ← ESP; FI;
    ELSE (* OperandSize = 16 *)
        Push(BP); (* RSP or (E)SP decrements by 2 *)
        FrameTemp ← SP;
FI;

IF NestingLevel = 0
    THEN GOTO CONTINUE;
FI;

IF (NestingLevel > 1)
    THEN FOR i ← 1 to (NestingLevel - 1)
        DO
            IF (OperandSize = 64)
                THEN
                    RBP ← RBP - 8;
                    Push([RBP]); (* Quadword push *)
                ELSE IF OperandSize = 32
                    THEN
                        IF StackSize = 32
                            THEN
                                EBP ← EBP - 4;
                                Push([EBP]); (* Doubleword push *)
                            ELSE (* StackSize = 16 *)
                                BP ← BP - 4;
                                Push([BP]); (* Doubleword push *)
                            FI;
                        FI;
                    ELSE (* OperandSize = 16 *)
                        IF StackSize = 32
                            THEN
                                EBP ← EBP - 2;
                                Push([EBP]); (* Word push *)
                            ELSE (* StackSize = 16 *)
                                BP ← BP - 2;
                                Push([BP]); (* Word push *)
                            FI;
                        FI;
                    FI;
            OD;
        FI;

IF (OperandSize = 64) (* nestinglevel 1 *)
    THEN
        Push(FrameTemp); (* Quadword push and RSP decrements by 8 *)
    ELSE IF OperandSize = 32

```

```

    THEN
        Push(FrameTemp); FI; (* Doubleword push and (E)SP decrements by 4 *)
    ELSE (* OperandSize = 16 *)
        Push(FrameTemp); (* Word push and RSP|ESP|SP decrements by 2 *)
FI;

CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
        RBP ← FrameTemp;
        RSP ← RSP – AllocSize;
    ELSE IF OperandSize = 32
        THEN
            EBP ← FrameTemp;
            ESP ← ESP – AllocSize; FI;
    ELSE (* OperandSize = 16 *)
        BP ← FrameTemp[15:1]; (* Bits 16 and above of applicable RBP/EBP are unmodified *)
        SP ← SP – AllocSize;
FI;

END;

```

**Flags Affected**

None.

**Protected Mode Exceptions**

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#SS	If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

## EXTRACTPS—Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm1, imm8	A	VV	SSE4_1	Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	A	V/V	AVX	Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
EVEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	B	V/V	AVX512F	Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

Extracts a single-precision floating-point value from the source operand (second operand) at the 32-bit offset specified from imm8. Immediate bits higher than the most significant offset for the vector length are ignored.

The extracted single-precision floating-point value is stored in the low 32-bits of the destination operand

In 64-bit mode, destination register operand has default operand size of 64 bits. The upper 32-bits of the register are filled with zero. REX.W is ignored.

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

VEX.vvvv/EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

If VEXTRACTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

### Operation

#### VEXTRACTPS (EVEX and VEX.128 encoded version)

SRC\_OFFSET ← IMM8[1:0]

IF (64-Bit Mode and DEST is register)

DEST[31:0] ← (SRC[127:0] >> (SRC\_OFFSET\*32)) AND 0FFFFFFFh

DEST[63:32] ← 0

ELSE

DEST[31:0] ← (SRC[127:0] >> (SRC\_OFFSET\*32)) AND 0FFFFFFFh

FI

**EXTRACTPS (128-bit Legacy SSE version)**

SRC\_OFFSET ← IMM8[1:0]

IF (64-Bit Mode and DEST is register)

DEST[31:0] ← (SRC[127:0] &gt;&gt; (SRC\_OFFSET\*32)) AND 0FFFFFFFh

DEST[63:32] ← 0

ELSE

DEST[31:0] ← (SRC[127:0] &gt;&gt; (SRC\_OFFSET\*32)) AND 0FFFFFFFh

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

EXTRACTPS int \_mm\_extract\_ps (\_\_m128 a, const int idx);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 5; Additionally

EVEX-encoded instructions, see Exceptions Type E9NF.

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## F2XM1—Compute $2^x-1$

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F0	F2XM1	Valid	Valid	Replace ST(0) with $(2^{\text{ST}(0)} - 1)$ .

### Description

Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range  $-1.0$  to  $+1.0$ . If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-16. Results Obtained from F2XM1**

ST(0) SRC	ST(0) DEST
$-1.0$ to $-0$	$-0.5$ to $-0$
$-0$	$-0$
$+0$	$+0$
$+0$ to $+1.0$	$+0$ to $1.0$

Values other than 2 can be exponentiated using the following formula:

$$x^y \leftarrow 2^{(y * \log_2 x)}$$

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$\text{ST}(0) \leftarrow (2^{\text{ST}(0)} - 1)$ ;

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow occurred.

#IA Source operand is an SNaN value or unsupported format.

#D Source is a denormal value.

#U Result is too small for destination format.

#P Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### **Compatibility Mode Exceptions**

Same exceptions as in protected mode.

### **64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FABS—Absolute Value

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 E1	FABS	Valid	Valid	Replace ST with its absolute value.

### Description

Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

**Table 3-17. Results Obtained from FABS**

ST(0) SRC	ST(0) DEST
$-\infty$	$+\infty$
$-F$	$+F$
$-0$	$+0$
$+0$	$+0$
$+F$	$+F$
$+\infty$	$+\infty$
NaN	NaN

#### NOTES:

F Means finite floating-point value.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$ST(0) \leftarrow |ST(0)|;$

### FPU Flags Affected

C1 Set to 0.  
C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow occurred.

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.



**FADD/FADDP/FIADD—Add**

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /0	FADD <i>m32fp</i>	Valid	Valid	Add <i>m32fp</i> to ST(0) and store result in ST(0).
DC /0	FADD <i>m64fp</i>	Valid	Valid	Add <i>m64fp</i> to ST(0) and store result in ST(0).
D8 C0+i	FADD ST(0), ST(i)	Valid	Valid	Add ST(0) to ST(i) and store result in ST(0).
DC C0+i	FADD ST(i), ST(0)	Valid	Valid	Add ST(i) to ST(0) and store result in ST(i).
DE C0+i	FADDP ST(i), ST(0)	Valid	Valid	Add ST(0) to ST(i), store result in ST(i), and pop the register stack.
DE C1	FADDP	Valid	Valid	Add ST(0) to ST(1), store result in ST(1), and pop the register stack.
DA /0	FIADD <i>m32int</i>	Valid	Valid	Add <i>m32int</i> to ST(0) and store result in ST(0).
DE /0	FIADD <i>m16int</i>	Valid	Valid	Add <i>m16int</i> to ST(0) and store result in ST(0).

**Description**

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward  $-\infty$  mode, in which case the result is  $-0$ . When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is  $\infty$  of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-18.

Table 3-18. FADD/FADDP/FIADD Results

SRC	DEST							
		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	NaN
	$-F$ or $-I$	$-\infty$	$-F$	SRC	SRC	$\pm F$ or $\pm 0$	$+\infty$	NaN
	$-0$	$-\infty$	DEST	$-0$	$\pm 0$	DEST	$+\infty$	NaN
	$+0$	$-\infty$	DEST	$\pm 0$	$+0$	DEST	$+\infty$	NaN
	$+F$ or $+I$	$-\infty$	$\pm F$ or $\pm 0$	SRC	SRC	$+F$	$+\infty$	NaN
	$+\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

F Means finite floating-point value.

I Means integer.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FIADD

THEN

DEST  $\leftarrow$  DEST + ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (\* Source operand is floating-point value \*)

DEST  $\leftarrow$  DEST + SRC;

FI;

IF Instruction = FADDP

THEN

PopRegisterStack;

FI;

**FPU Flags Affected**

- C1 Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.
- C0, C2, C3 Undefined.

**Floating-Point Exceptions**

- #IS Stack underflow occurred.
- #IA Operand is an SNaN value or unsupported format.  
Operands are infinities of unlike sign.
- #D Source operand is a denormal value.
- #U Result is too small for destination format.
- #O Result is too large for destination format.
- #P Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FBLD—Load Binary Coded Decimal

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /4	FBLD <i>m80dec</i>	Valid	Valid	Convert BCD value to floating-point and push onto the FPU stack.

### Description

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of  $-0$ .

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$TOP \leftarrow TOP - 1$ ;

$ST(0) \leftarrow \text{ConvertToDoubleExtendedPrecisionFP}(\text{SRC})$ ;

### FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, set to 0.  
C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack overflow occurred.

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
If the DS, ES, FS, or GS register contains a NULL segment selector.  
#SS(0) If a memory operand effective address is outside the SS segment limit.  
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#PF(fault-code) If a page fault occurs.  
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.  
#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
#SS If a memory operand effective address is outside the SS segment limit.  
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
#SS(0) If a memory operand effective address is outside the SS segment limit.  
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#PF(fault-code) If a page fault occurs.  
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.  
#UD If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FBSTP—Store BCD Integer and Pop

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /6	FBSTP m80bcd	Valid	Valid	Store ST(0) in m80bcd and pop ST(0).

### Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

**Table 3-19. FBSTP Results**

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	$-D$
$-1 < F < -0$	**
$-0$	$-0$
$+0$	$+0$
$+0 < F < +1$	**
$F \geq +1$	$+D$
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

#### NOTES:

F Means finite floating-point value.

D Means packed-BCD number.

\* Indicates floating-point invalid-operation (#IA) exception.

\*\*  $\pm 0$  or  $\pm 1$ , depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an  $\infty$ , SNaN, QNaN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$DEST \leftarrow BCD(ST(0));$

PopRegisterStack;

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Converted value that exceeds 18 BCD digits in length. Source operand is an SNaN, QNaN, $\pm\infty$ , or in an unsupported format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#GP(0)	If a segment register is being loaded with a segment selector that points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FCFS—Change Sign

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 E0	FCFS	Valid	Valid	Complements sign of ST(0).

### Description

Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa. The following table shows the results obtained when changing the sign of various classes of numbers.

**Table 3-20. FCFS Results**

ST(0) SRC	ST(0) DEST
$-\infty$	$+\infty$
$-F$	$+F$
$-0$	$+0$
$+0$	$-0$
$+F$	$-F$
$+\infty$	$-\infty$
NaN	NaN

#### NOTES:

\* F means finite floating-point value.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$\text{SignBit}(\text{ST}(0)) \leftarrow \text{NOT}(\text{SignBit}(\text{ST}(0)))$ ;

### FPU Flags Affected

C1 Set to 0.  
C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow occurred.

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.



### **64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FCLEX/FNCLEX—Clear Exceptions

Opcode*	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DB E2	FCLEX	Valid	Valid	Clear floating-point exception flags after checking for pending unmasked floating-point exceptions.
DB E2	FNCLEX*	Valid	Valid	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

### IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS\* compatibility mode, it is possible (under unusual circumstances) for an FNCLEX instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNCLEX instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCRS register.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```
FPUStatusWord[0:7] ← 0;
FPUStatusWord[15] ← 0;
```

### FPU Flags Affected

The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

### Floating-Point Exceptions

None

### Protected Mode Exceptions

```
#NM          CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD          If the LOCK prefix is used.
```

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### **Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

### **Compatibility Mode Exceptions**

Same exceptions as in protected mode.

### **64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FCMOVcc—Floating-Point Conditional Move

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode*	Description
DA C0+i	FCMOVB ST(0), ST(i)	Valid	Valid	Move if below (CF=1).
DA C8+i	FCMOVE ST(0), ST(i)	Valid	Valid	Move if equal (ZF=1).
DA D0+i	FCMOVBE ST(0), ST(i)	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
DA D8+i	FCMOVU ST(0), ST(i)	Valid	Valid	Move if unordered (PF=1).
DB C0+i	FCMOVNB ST(0), ST(i)	Valid	Valid	Move if not below (CF=0).
DB C8+i	FCMOVNE ST(0), ST(i)	Valid	Valid	Move if not equal (ZF=0).
DB D0+i	FCMOVNBE ST(0), ST(i)	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
DB D8+i	FCMOVNU ST(0), ST(i)	Valid	Valid	Move if not unordered (PF=0).

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The condition for each mnemonic is given in the Description column above and in Chapter 8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. The source operand is always in the ST(i) register and the destination operand is always ST(0).

The FCMOVcc instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor may not support the FCMOVcc instructions. Software can check if the FCMOVcc instructions are supported by checking the processor's feature information with the CUID instruction (see "COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS" in this chapter). If both the CMOV and FPU feature bits are set, the FCMOVcc instructions are supported.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

The FCMOVcc instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

### Operation

IF condition TRUE  
 THEN ST(0) ← ST(i);  
 FI;

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred.  
 C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow occurred.

### Integer Flags Affected

None.

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FCOM/FCOMP/FCOMPP—Compare Floating Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /2	FCOM <i>m32fp</i>	Valid	Valid	Compare ST(0) with <i>m32fp</i> .
DC /2	FCOM <i>m64fp</i>	Valid	Valid	Compare ST(0) with <i>m64fp</i> .
D8 D0+i	FCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
D8 D1	FCOM	Valid	Valid	Compare ST(0) with ST(1).
D8 /3	FCOMP <i>m32fp</i>	Valid	Valid	Compare ST(0) with <i>m32fp</i> and pop register stack.
DC /3	FCOMP <i>m64fp</i>	Valid	Valid	Compare ST(0) with <i>m64fp</i> and pop register stack.
D8 D8+i	FCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

### Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that  $-0.0$  is equal to  $+0.0$ .

**Table 3-21. FCOM/FCOMP/FCOMPP Results**

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered*	1	1	1

#### NOTES:

\* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to “unordered.” If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## Operation

CASE (relation of operands) OF

ST > SRC: C3, C2, C0 ← 000;

ST < SRC: C3, C2, C0 ← 001;

ST = SRC: C3, C2, C0 ← 100;

ESAC;

IF ST(0) or SRC = NaN or unsupported format

THEN

#IA

IF FPUControlWord.IM = 1

THEN

C3, C2, C0 ← 111;

FI;

FI;

IF Instruction = FCOMP

THEN

PopRegisterStack;

FI;

IF Instruction = FCOMPP

THEN

PopRegisterStack;

PopRegisterStack;

FI;

## FPU Flags Affected

C1 Set to 0.

C0, C2, C3 See table on previous page.

## Floating-Point Exceptions

#IS Stack underflow occurred.

#IA One or both operands are NaN values or have unsupported formats.

Register is marked empty.

#D One or both operands are denormal values.

## Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.



## FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DB F0+i	FCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i) and set status flags accordingly.
DF F0+i	FCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), set status flags accordingly, and pop register stack.
DB E8+i	FUCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly.
DF E8+i	FUCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack.

### Description

Performs an unordered comparison of the contents of registers ST(0) and ST(i) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that  $-0.0$  is equal to  $+0.0$ .

**Table 3-22. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results**

Comparison Results*	ZF	PF	CF
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered**	1	1	1

#### NOTES:

\* See the IA-32 Architecture Compatibility section below.

\*\* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). The FUCOMI/FUCOMIP instructions perform the same operations as the FCOMI/FCOMIP instructions. The only difference is that the FUCOMI/FUCOMIP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOMI/FCOMIP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

If the operation results in an invalid-arithmetic-operand exception being raised, the status flags in the EFLAGS register are set only if the exception is masked.

The FCOMI/FCOMIP and FUCOMI/FUCOMIP instructions set the OF, SF and AF flags to zero in the EFLAGS register (regardless of whether an invalid-operation exception is detected).

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

## Operation

CASE (relation of operands) OF

ST(0) > ST(i): ZF, PF, CF ← 000;

ST(0) < ST(i): ZF, PF, CF ← 001;

ST(0) = ST(i): ZF, PF, CF ← 100;

ESAC;

IF Instruction is FCOMI or FCOMIP

THEN

IF ST(0) or ST(i) = NaN or unsupported format

THEN

#IA

IF FPUControlWord.IM = 1

THEN

ZF, PF, CF ← 111;

FI;

FI;

FI;

IF Instruction is FUCOMI or FUCOMIP

THEN

IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format

THEN

ZF, PF, CF ← 111;

ELSE (\* ST(0) or ST(i) is SNaN or unsupported format \*)

#IA;

IF FPUControlWord.IM = 1

THEN

ZF, PF, CF ← 111;

FI;

FI;

FI;

IF Instruction is FCOMIP or FUCOMIP

THEN

PopRegisterStack;

FI;

## FPU Flags Affected

C1 Set to 0.

C0, C2, C3 Not affected.

## Floating-Point Exceptions

#IS Stack underflow occurred.

#IA (FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats.  
(FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception.

**Protected Mode Exceptions**

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FCOS— Cosine

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FF	FCOS	Valid	Valid	Replace ST(0) with its approximate cosine.

### Description

Computes the approximate cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range  $-2^{63}$  to  $+2^{63}$ . The following table shows the results obtained when taking the cosine of various classes of numbers.

**Table 3-23. FCOS Results**

ST(0) SRC	ST(0) DEST
$-\infty$	*
$-F$	$-1$ to $+1$
$-0$	$+1$
$+0$	$+1$
$+F$	$-1$ to $+1$
$+\infty$	*
NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range  $-2^{63}$  to  $+2^{63}$  can be reduced to the range of the instruction by subtracting an appropriate integer multiple of  $2\pi$ . However, even within the range  $-2^{63}$  to  $+2^{63}$ , inaccurate results can occur because the finite approximation of  $\pi$  used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FCOS only to arguments reduced accurately in software, to a value smaller in absolute value than  $3\pi/8$ . See the sections titled “Approximation of Pi” and “Transcendental Instruction Accuracy” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a discussion of the proper value to use for  $\pi$  in performing such reductions.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

IF  $|ST(0)| < 2^{63}$

THEN

C2  $\leftarrow$  0;

ST(0)  $\leftarrow$  FCOS(ST(0)); // approximation of cosine

ELSE (\* Source operand is out-of-range \*)

C2  $\leftarrow$  1;

FI;

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise. Undefined if C2 is 1.
C2	Set to 1 if outside range ( $-2^{63} < \text{source operand} < +2^{63}$ ); otherwise, set to 0.
C0, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , or unsupported format.
#D	Source is a denormal value.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FDECSTP—Decrement Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F6	FDECSTP	Valid	Valid	Decrement TOP field in FPU status word.

### Description

Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). If the TOP field contains a 0, it is set to 7. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```
IF TOP = 0
    THEN TOP ← 7;
    ELSE TOP ← TOP - 1;
FI;
```

### FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
 #MF If there is a pending x87 FPU exception.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

**FDIV/FDIVP/FIDIV—Divide**

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /6	FDIV <i>m32fp</i>	Valid	Valid	Divide ST(0) by <i>m32fp</i> and store result in ST(0).
DC /6	FDIV <i>m64fp</i>	Valid	Valid	Divide ST(0) by <i>m64fp</i> and store result in ST(0).
D8 F0+i	FDIV ST(0), ST(i)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(0).
DC F8+i	FDIV ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(i).
DE F8+i	FDIVP ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE F9	FDIVP	Valid	Valid	Divide ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /6	FIDIV <i>m32int</i>	Valid	Valid	Divide ST(0) by <i>m32int</i> and store result in ST(0).
DE /6	FIDIV <i>m16int</i>	Valid	Valid	Divide ST(0) by <i>m16int</i> and store result in ST(0).

**Description**

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(i) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to double extended-precision floating-point format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an  $\infty$  of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-24. FDIV/FDIVP/FIDIV Results

		DEST						
SRC		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	*	$+0$	$+0$	$-0$	$-0$	*	NaN
	$-F$	$+\infty$	$+F$	$+0$	$-0$	$-F$	$-\infty$	NaN
	$-I$	$+\infty$	$+F$	$+0$	$-0$	$-F$	$-\infty$	NaN
	$-0$	$+\infty$	**	*	*	**	$-\infty$	NaN
	$+0$	$-\infty$	**	*	*	**	$+\infty$	NaN
	$+I$	$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$+F$	$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$+\infty$	*	$-0$	$-0$	$+0$	$+0$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

F Means finite floating-point value.

I Means integer.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

\*\* Indicates floating-point zero-divide (#Z) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

```

IF SRC = 0
    THEN
        #Z;
    ELSE
        IF Instruction is FIDIV
            THEN
                DEST ← DEST / ConvertToDoubleExtendedPrecisionFP(SRC);
            ELSE (* Source operand is floating-point value *)
                DEST ← DEST / SRC;
        FI;
    FI;

```

```

IF Instruction = FDIVP
    THEN
        PopRegisterStack;
    FI;

```

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.



**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format. $\pm\infty / \pm\infty$ ; $\pm 0 / \pm 0$
#D	Source is a denormal value.
#Z	DEST / $\pm 0$ , where DEST is not equal to $\pm 0$ .
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FDIVR/FDIVRP/FIDIVR—Reverse Divide

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /7	FDIVR <i>m32fp</i>	Valid	Valid	Divide <i>m32fp</i> by ST(0) and store result in ST(0).
DC /7	FDIVR <i>m64fp</i>	Valid	Valid	Divide <i>m64fp</i> by ST(0) and store result in ST(0).
D8 F8+i	FDIVR ST(0), ST(i)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(0).
DC F0+i	FDIVR ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(i).
DE F0+i	FDIVRP ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i), store result in ST(i), and pop the register stack.
DE F1	FDIVRP	Valid	Valid	Divide ST(0) by ST(1), store result in ST(1), and pop the register stack.
DA /7	FIDIVR <i>m32int</i>	Valid	Valid	Divide <i>m32int</i> by ST(0) and store result in ST(0).
DE /7	FIDIVR <i>m16int</i>	Valid	Valid	Divide <i>m16int</i> by ST(0) and store result in ST(0).

### Description

Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a floating-point or an integer value) by the contents of the ST(0) register. The two-operand version, divides the contents of the ST(i) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to double extended-precision floating-point format before performing the division.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an  $\infty$  of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-25. FDIVR/FDIVRP/FIDIVR Results

		DEST						
SRC		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	*	$+\infty$	$+\infty$	$-\infty$	$-\infty$	*	NaN
	$-F$	$+0$	$+F$	**	**	$-F$	$-0$	NaN
	$-I$	$+0$	$+F$	**	**	$-F$	$-0$	NaN
	$-0$	$+0$	$+0$	*	*	$-0$	$-0$	NaN
	$+0$	$-0$	$-0$	*	*	$+0$	$+0$	NaN
	$+I$	$-0$	$-F$	**	**	$+F$	$+0$	NaN
	$+F$	$-0$	$-F$	**	**	$+F$	$+0$	NaN
	$+\infty$	*	$-\infty$	$-\infty$	$+\infty$	$+\infty$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

- F Means finite floating-point value.
- I Means integer.
- \* Indicates floating-point invalid-arithmetic-operand (#IA) exception.
- \*\* Indicates floating-point zero-divide (#Z) exception.

When the source operand is an integer 0, it is treated as a +0. This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
IF DEST = 0
    THEN
        #Z;
    ELSE
        IF Instruction = FIDIVR
            THEN
                DEST ← ConvertToDoubleExtendedPrecisionFP(SRC) / DEST;
            ELSE (* Source operand is floating-point value *)
                DEST ← SRC / DEST;
        FI;
    FI;

IF Instruction = FDIVRP
    THEN
        PopRegisterStack;
    FI;
```

FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.
- C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format. $\pm\infty / \pm\infty$ ; $\pm 0 / \pm 0$
#D	Source is a denormal value.
#Z	$SRC / \pm 0$ , where SRC is not equal to $\pm 0$ .
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FFREE—Free Floating-Point Register

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DD C0+i	FFREE ST(i)	Valid	Valid	Sets tag for ST(i) to empty.

### Description

Sets the tag in the FPU tag register associated with register ST(i) to empty (11B). The contents of ST(i) and the FPU stack-top pointer (TOP) are not affected.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$TAG(i) \leftarrow 11B;$

### FPU Flags Affected

C0, C1, C2, C3 undefined.

### Floating-Point Exceptions

None

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
 #MF If there is a pending x87 FPU exception.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FICOM/FICOMP—Compare Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DE /2	FICOM <i>m16int</i>	Valid	Valid	Compare ST(0) with <i>m16int</i> .
DA /2	FICOM <i>m32int</i>	Valid	Valid	Compare ST(0) with <i>m32int</i> .
DE /3	FICOMP <i>m16int</i>	Valid	Valid	Compare ST(0) with <i>m16int</i> and pop stack register.
DA /3	FICOMP <i>m32int</i>	Valid	Valid	Compare ST(0) with <i>m32int</i> and pop stack register.

### Description

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to double extended-precision floating-point format before the comparison is made.

**Table 3-26. FICOM/FICOMP Results**

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered	1	1	1

These instructions perform an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). If either operand is a NaN or is in an undefined format, the condition flags are set to “unordered.”

The sign of zero is ignored, so that  $-0.0 \leftarrow +0.0$ .

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

CASE (relation of operands) OF

ST(0) > SRC: C3, C2, C0  $\leftarrow$  000;  
 ST(0) < SRC: C3, C2, C0  $\leftarrow$  001;  
 ST(0) = SRC: C3, C2, C0  $\leftarrow$  100;  
 Unordered: C3, C2, C0  $\leftarrow$  111;

ESAC;

IF Instruction = FICOMP

THEN

PopRegisterStack;

FI;

### FPU Flags Affected

C1 Set to 0.  
 C0, C2, C3 See table on previous page.

### Floating-Point Exceptions

#IS Stack underflow occurred.  
 #IA One or both operands are NaN values or have unsupported formats.  
 #D One or both operands are denormal values.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FILD—Load Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DF /0	FILD <i>m16int</i>	Valid	Valid	Push <i>m16int</i> onto the FPU register stack.
DB /0	FILD <i>m32int</i>	Valid	Valid	Push <i>m32int</i> onto the FPU register stack.
DF /5	FILD <i>m64int</i>	Valid	Valid	Push <i>m64int</i> onto the FPU register stack.

### Description

Converts the signed-integer source operand into double extended-precision floating-point format and pushes the value onto the FPU register stack. The source operand can be a word, doubleword, or quadword integer. It is loaded without rounding errors. The sign of the source operand is preserved.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

TOP ← TOP – 1;  
ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);

### FPU Flags Affected

C1 Set to 1 if stack overflow occurred; set to 0 otherwise.  
C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack overflow occurred.

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
If the DS, ES, FS, or GS register contains a NULL segment selector.  
#SS(0) If a memory operand effective address is outside the SS segment limit.  
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#PF(fault-code) If a page fault occurs.  
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.  
#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
#SS If a memory operand effective address is outside the SS segment limit.  
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
#SS(0) If a memory operand effective address is outside the SS segment limit.  
#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#PF(fault-code) If a page fault occurs.  
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.  
#UD If the LOCK prefix is used.



Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FINCSTP—Increment Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F7	FINCSTP	Valid	Valid	Increment the TOP field in the FPU status register.

### Description

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). If the TOP field contains a 7, it is set to 0. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```
IF TOP = 7
    THEN TOP ← 0;
    ELSE TOP ← TOP + 1;
FI;
```

### FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

### Floating-Point Exceptions

None

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
 #MF If there is a pending x87 FPU exception.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FINIT/FNINIT—Initialize Floating-Point Unit

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DB E3	FINIT	Valid	Valid	Initialize FPU after checking for pending unmasked floating-point exceptions.
DB E3	FNINIT*	Valid	Valid	Initialize FPU without checking for pending unmasked floating-point exceptions.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

The assembler issues two instructions for the FINIT instruction (an FWAIT instruction followed by an FNINIT instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNINIT instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNINIT instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

In the Intel387 math coprocessor, the FINIT/FNINIT instruction does not clear the instruction and data pointers.

This instruction affects only the x87 FPU. It does not affect the XMM and MXCSR registers.

### Operation

```

FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;

```

### FPU Flags Affected

C0, C1, C2, C3 set to 0.

### Floating-Point Exceptions

None

### Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FIST/FISTP—Store Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DF /2	FIST <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> .
DB /2	FIST <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> .
DF /3	FISTP <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> and pop register stack.
DB /3	FISTP <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> and pop register stack.
DF /7	FISTP <i>m64int</i>	Valid	Valid	Store ST(0) in <i>m64int</i> and pop register stack.

### Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word or doubleword integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction also stores values in quadword integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

**Table 3-27. FIST/FISTP Results**

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	$-I$
$-1 < F < -0$	**
$-0$	0
$+0$	0
$+0 < F < +1$	**
$F \geq +1$	$+I$
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

**NOTES:**  
 F Means finite floating-point value.  
 I Means integer.  
 \* Indicates floating-point invalid-operation (#IA) exception.  
 \*\* 0 or  $\pm 1$ , depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the converted value is too large for the destination format, or if the source operand is an  $\infty$ , SNaN, QNaN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in memory.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

DEST  $\leftarrow$  Integer(ST(0));

IF Instruction = FISTP  
 THEN  
   PopRegisterStack;  
 FI;

**FPU Flags Affected**

C1                   Set to 0 if stack underflow occurred.  
                     Indicates rounding direction of if the inexact exception (#P) is generated: 0  $\leftarrow$  not roundup; 1  $\leftarrow$  roundup.  
                     Set to 0 otherwise.  
 C0, C2, C3          Undefined.

**Floating-Point Exceptions**

#IS                  Stack underflow occurred.  
 #IA                  Converted value is too large for the destination format.  
                     Source operand is an SNaN, QNaN,  $\pm\infty$ , or unsupported format.  
 #P                  Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#GP(0)              If the destination is located in a non-writable segment.  
                     If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
                     If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.  
 #SS(0)              If a memory operand effective address is outside the SS segment limit.  
 #NM                  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
 #PF(fault-code)    If a page fault occurs.  
 #AC(0)              If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.  
 #UD                  If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP                  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
 #SS                  If a memory operand effective address is outside the SS segment limit.  
 #NM                  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
 #UD                  If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)              If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
 #SS(0)              If a memory operand effective address is outside the SS segment limit.  
 #NM                  CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
 #PF(fault-code)    If a page fault occurs.  
 #AC(0)              If alignment checking is enabled and an unaligned memory reference is made.  
 #UD                  If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FISTTP—Store Integer with Truncation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /1	FISTTP <i>m16int</i>	Valid	Valid	Store ST(0) in <i>m16int</i> with truncation.
DB /1	FISTTP <i>m32int</i>	Valid	Valid	Store ST(0) in <i>m32int</i> with truncation.
DD /1	FISTTP <i>m64int</i>	Valid	Valid	Store ST(0) in <i>m64int</i> with truncation.

### Description

FISTTP converts the value in ST into a signed integer using truncation (chop) as rounding mode, transfers the result to the destination, and pop ST. FISTTP accepts word, short integer, and long integer destinations.

The following table shows the results obtained when storing various classes of numbers in integer format.

**Table 3-28. FISTTP Results**

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	$-I$
$-1 < F < +1$	0
$F \geq +1$	$+I$
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

#### NOTES:

F Means finite floating-point value.

I Means integer.

\* Indicates floating-point invalid-operation (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

DEST  $\leftarrow$  ST;

pop ST;

### Flags Affected

C1 is cleared; C0, C2, C3 undefined.

### Numeric Exceptions

Invalid, Stack Invalid (stack underflow), Precision.

### Protected Mode Exceptions

#GP(0)	If the destination is in a nonwritable segment.
	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#NM	If CR0.EM[bit 2] = 1.
	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.SSE3[bit 0] = 0.
	If the LOCK prefix is used.



**Real Address Mode Exceptions**

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.EM[bit 2] = 1. If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.SSE3[bit 0] = 0. If the LOCK prefix is used.

**Virtual 8086 Mode Exceptions**

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.EM[bit 2] = 1. If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.SSE3[bit 0] = 0. If the LOCK prefix is used.
#PF(fault-code)	For a page fault.
#AC(0)	For unaligned memory reference if the current privilege is 3.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. If the LOCK prefix is used.

## FLD—Load Floating Point Value

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 /0	FLD <i>m32fp</i>	Valid	Valid	Push <i>m32fp</i> onto the FPU register stack.
DD /0	FLD <i>m64fp</i>	Valid	Valid	Push <i>m64fp</i> onto the FPU register stack.
DB /5	FLD <i>m80fp</i>	Valid	Valid	Push <i>m80fp</i> onto the FPU register stack.
D9 C0+i	FLD ST(i)	Valid	Valid	Push ST(i) onto the FPU register stack.

### Description

Pushes the source operand onto the FPU register stack. The source operand can be in single-precision, double-precision, or double extended-precision floating-point format. If the source operand is in single-precision or double-precision floating-point format, it is automatically converted to the double extended-precision floating-point format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(i)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```

IF SRC is ST(i)
    THEN
        temp ← ST(i);
FI;

TOP ← TOP – 1;

IF SRC is memory-operand
    THEN
        ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* SRC is ST(i) *)
        ST(0) ← temp;
FI;
```

### FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, set to 0.  
C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow or overflow occurred.  
#IA Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD *m80fp* or FLD ST(i)).  
#D Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.

**Protected Mode Exceptions**

#GP(0)	If destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E8	FLD1	Valid	Valid	Push +1.0 onto the FPU register stack.
D9 E9	FLDL2T	Valid	Valid	Push $\log_2 10$ onto the FPU register stack.
D9 EA	FLDL2E	Valid	Valid	Push $\log_2 e$ onto the FPU register stack.
D9 EB	FLDPI	Valid	Valid	Push $\pi$ onto the FPU register stack.
D9 EC	FLDLG2	Valid	Valid	Push $\log_{10} 2$ onto the FPU register stack.
D9 ED	FLDLN2	Valid	Valid	Push $\log_e 2$ onto the FPU register stack.
D9 EE	FLDZ	Valid	Valid	Push +0.0 onto the FPU register stack.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Push one of seven commonly used constants (in double extended-precision floating-point format) onto the FPU register stack. The constants that can be loaded with these instructions include +1.0, +0.0,  $\log_2 10$ ,  $\log_2 e$ ,  $\pi$ ,  $\log_{10} 2$ , and  $\log_e 2$ . For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to double extended-precision floating-point format. The inexact-result exception (#P) is not generated as a result of the rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up.

See the section titled “Approximation of Pi” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the  $\pi$  constant.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel 287 math coprocessors.

### Operation

TOP  $\leftarrow$  TOP – 1;  
ST(0)  $\leftarrow$  CONSTANT;

### FPU Flags Affected

C1 Set to 1 if stack overflow occurred; otherwise, set to 0.  
C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack overflow occurred.

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#MF If there is a pending x87 FPU exception.  
#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### **Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

### **Compatibility Mode Exceptions**

Same exceptions as in protected mode.

### **64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FLDCW—Load x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /5	FLDCW m2byte	Valid	Valid	Load FPU control word from <i>m2byte</i> .

### Description

Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU's mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmask one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

FPUControlWord ← SRC;

### FPU Flags Affected

C0, C1, C2, C3 undefined.

### Floating-Point Exceptions

None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next "waiting" floating-point instruction.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FLDENV—Load x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 /4	FLDENV <i>m14/28byte</i>	Valid	Valid	Load FPU environment from <i>m14byte</i> or <i>m28byte</i> .

### Description

Loads the complete x87 FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory. This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled “Software Exception Handling” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

If a page or limit fault occurs during the execution of this instruction, the state of the x87 FPU registers as seen by the fault handler may be different than the state being loaded from memory. In such situations, the fault handler should ignore the status of the x87 FPU registers, handle the fault, and return. The FLDENV instruction will then complete the loading of the x87 FPU registers with no resulting context inconsistency.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```

FPUControlWord ← SRC[FPUControlWord];
FPUStatusWord ← SRC[FPUStatusWord];
FPUTagWord ← SRC[FPUTagWord];
FPUDataPointer ← SRC[FPUDataPointer];
FPUInstructionPointer ← SRC[FPUInstructionPointer];
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode];

```

### FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

### Floating-Point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next “waiting” floating-point instruction.



**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FMUL/FMULP/FIMUL—Multiply

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /1	FMUL <i>m32fp</i>	Valid	Valid	Multiply ST(0) by <i>m32fp</i> and store result in ST(0).
DC /1	FMUL <i>m64fp</i>	Valid	Valid	Multiply ST(0) by <i>m64fp</i> and store result in ST(0).
D8 C8+i	FMUL ST(0), ST(i)	Valid	Valid	Multiply ST(0) by ST(i) and store result in ST(0).
DC C8+i	FMUL ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0) and store result in ST(i).
DE C8+i	FMULP ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE C9	FMULP	Valid	Valid	Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /1	FIMUL <i>m32int</i>	Valid	Valid	Multiply ST(0) by <i>m32int</i> and store result in ST(0).
DE /1	FIMUL <i>m16int</i>	Valid	Valid	Multiply ST(0) by <i>m16int</i> and store result in ST(0).

### Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be an FPU data register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction multiplies the contents of the ST(1) register by the contents of the ST(0) register and stores the product in the ST(1) register. The one-operand version multiplies the contents of the ST(0) register by the contents of a memory location (either a floating point or an integer value) and stores the product in the ST(0) register. The two-operand version, multiplies the contents of the ST(0) register by the contents of the ST(i) register, or vice versa, with the result being stored in the register specified with the first operand (the destination operand).

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to double extended-precision floating-point format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or  $\infty$ . When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-29. FMUL/FMULP/FIMUL Results

SRC	DEST						
		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$
	$-\infty$	$+\infty$	$+\infty$	*	*	$-\infty$	$-\infty$
	$-F$	$+\infty$	$+F$	$+0$	$-0$	$-F$	$-\infty$
	$-I$	$+\infty$	$+F$	$+0$	$-0$	$-F$	$-\infty$
	$-0$	*	$+0$	$+0$	$-0$	$-0$	*
	$+0$	*	$-0$	$-0$	$+0$	$+0$	*
	$+I$	$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$
	$+F$	$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$
	$+\infty$	$-\infty$	$-\infty$	*	*	$+\infty$	$+\infty$
	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

F Means finite floating-point value.

I Means Integer.

\* Indicates invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FIMUL

THEN

DEST  $\leftarrow$  DEST \* ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (\* Source operand is floating-point value \*)

DEST  $\leftarrow$  DEST \* SRC;

FI;

IF Instruction = FMULP

THEN

PopRegisterStack;

FI;

**FPU Flags Affected**

- C1 Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.
- C0, C2, C3 Undefined.

**Floating-Point Exceptions**

- #IS Stack underflow occurred.
- #IA Operand is an SNaN value or unsupported format.  
One operand is  $\pm 0$  and the other is  $\pm\infty$ .
- #D Source operand is a denormal value.
- #U Result is too small for destination format.
- #O Result is too large for destination format.
- #P Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FNOP—No Operation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 D0	FNOP	Valid	Valid	No operation is performed.

### Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register and the FPU Instruction Pointer.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### FPU Flags Affected

C0, C1, C2, C3 undefined.

### Floating-Point Exceptions

None

### Protected Mode Exceptions

- #NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
- #MF If there is a pending x87 FPU exception.
- #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FPATAN—Partial Arctangent

Opcode*	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F3	FPATAN	Valid	Valid	Replace ST(1) with $\arctan(\text{ST}(1)/\text{ST}(0))$ and pop the register stack.

NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than  $+\pi$ .

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point (−X,Y) is in the second quadrant, resulting in an angle between  $\pi/2$  and  $\pi$ , while a point (X,−Y) is in the fourth quadrant, resulting in an angle between 0 and  $-\pi/2$ . A point (−X,−Y) is in the third quadrant, giving an angle between  $-\pi/2$  and  $-\pi$ .

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

**Table 3-30. FPATAN Results**

		ST(0)						
ST(1)		$-\infty$	− F	− 0	+ 0	+ F	$+\infty$	NaN
	$-\infty$	$-3\pi/4^*$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/4^*$	NaN
	− F	−p	$-\pi$ to $-\pi/2$	$-\pi/2$	$-\pi/2$	$-\pi/2$ to $-0$	− 0	NaN
	− 0	−p	−p	$-p^*$	$-0^*$	− 0	− 0	NaN
	+ 0	+p	+ p	$+\pi^*$	$+0^*$	+ 0	+ 0	NaN
	+ F	+p	$+\pi$ to $+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$ to $+0$	+ 0	NaN
	$+\infty$	$+3\pi/4^*$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/2$	$+\pi/4^*$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

\* Table 8-10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, specifies that the ratios 0/0 and  $\infty/\infty$  generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the floating-point QNaN indefinite value is returned. With the FPATAN instruction, the 0/0 or  $\infty/\infty$  value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation,  $\arctan(0,0)$  etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow floating-point values as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$$0 \leq |\text{ST}(1)| < |\text{ST}(0)| < +\infty$$

**Operation**

$$ST(1) \leftarrow \arctan(ST(1) / ST(0));$$

PopRegisterStack;

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FPREM—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F8	FPREM	Valid	Valid	Replace ST(0) with the remainder obtained from dividing ST(0) by ST(1).

### Description

Computes the remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

$$\text{Remainder} \leftarrow \text{ST}(0) - (Q * \text{ST}(1))$$

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of  $[\text{ST}(0) / \text{ST}(1)]$  toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the inexact-result exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

**Table 3-31. FPREM Results**

		ST(1)						
ST(0)		$-\infty$	-F	-0	+0	+F	$+\infty$	NaN
	$-\infty$	*	*	*	*	*	*	NaN
	-F	ST(0)	-F or -0	**	**	-F or -0	ST(0)	NaN
	-0	-0	-0	*	*	-0	-0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	+F or +0	**	**	+F or +0	ST(0)	NaN
	$+\infty$	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

\*\* Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is  $\infty$ , the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name “partial remainder” because of the way it computes the remainder. This instruction arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU



status word. This information is important in argument reduction for the tangent function (using a modulus of  $\pi/4$ ), because it locates the original angle in the correct one of eight sectors of the unit circle.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```

D ← exponent(ST(0)) - exponent(ST(1));
IF D < 64
    THEN
        Q ← Integer(TruncateTowardZero(ST(0) / ST(1)));
        ST(0) ← ST(0) - (ST(1) * Q);
        C2 ← 0;
        C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 ← 1;
        N ← An implementation-dependent number between 32 and 63;
        QQ ← Integer(TruncateTowardZero((ST(0) / ST(1)) / 2(D-N)));
        ST(0) ← ST(0) - (ST(1) * QQ * 2(D-N));
FI;

```

### FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus is 0, dividend is $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

### Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FPREM1—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F5	FPREM1	Valid	Valid	Replace ST(0) with the IEEE remainder obtained from dividing ST(0) by ST(1).

### Description

Computes the IEEE remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

$$\text{Remainder} \leftarrow \text{ST}(0) - (Q * \text{ST}(1))$$

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of  $[\text{ST}(0) / \text{ST}(1)]$  toward the nearest integer value. The magnitude of the remainder is less than or equal to half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

**Table 3-32. FPREM1 Results**

		ST(1)						
ST(0)		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	*	*	*	*	*	*	NaN
	$-F$	ST(0)	$\pm F$ or $-0$	**	**	$\pm F$ or $-0$	ST(0)	NaN
	$-0$	$-0$	$-0$	*	*	$-0$	$-0$	NaN
	$+0$	$+0$	$+0$	*	*	$+0$	$+0$	NaN
	$+F$	ST(0)	$\pm F$ or $+0$	**	**	$\pm F$ or $+0$	ST(0)	NaN
	$+\infty$	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

\*\* Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is  $\infty$ , the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Standard 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the “Operation” section below).

Like the FPREM instruction, FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU

status word. This information is important in argument reduction for the tangent function (using a modulus of  $\pi/4$ ), because it locates the original angle in the correct one of eight sectors of the unit circle.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```

D ← exponent(ST(0)) - exponent(ST(1));
IF D < 64
    THEN
        Q ← Integer(RoundTowardNearestInteger(ST(0) / ST(1)));
        ST(0) ← ST(0) - (ST(1) * Q);
        C2 ← 0;
        C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 ← 1;
        N ← An implementation-dependent number between 32 and 63;
        QQ ← Integer(TruncateTowardZero((ST(0) / ST(1)) / 2(D-N)));
        ST(0) ← ST(0) - (ST(1) * QQ * 2(D-N));
FI;

```

### FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus (divisor) is 0, dividend is $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

### Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FPTAN—Partial Tangent

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F2	FPTAN	Valid	Valid	Replace ST(0) with its approximate tangent and push 1 onto the FPU stack.

### Description

Computes the approximate tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than  $\pm 2^{63}$ . The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

**Table 3-33. FPTAN Results**

ST(0) SRC	ST(0) DEST
$-\infty$	*
$-F$	$-F$ to $+F$
$-0$	$-0$
$+0$	$+0$
$+F$	$-F$ to $+F$
$+\infty$	*
NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range  $-2^{63}$  to  $+2^{63}$  can be reduced to the range of the instruction by subtracting an appropriate integer multiple of  $2\pi$ . However, even within the range  $-2^{63}$  to  $+2^{63}$ , inaccurate results can occur because the finite approximation of  $\pi$  used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FPTAN only to arguments reduced accurately in software, to a value smaller in absolute value than  $3\pi/8$ . See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for  $\pi$  in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

```

IF ST(0) < 263
  THEN
    C2 ← 0;
    ST(0) ← fptan(ST(0)); // approximation of tan
    TOP ← TOP – 1;
    ST(0) ← 1.0;
  ELSE (* Source operand is out-of-range *)
    C2 ← 1;
FI;

```

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range ( $-2^{63} < \text{source operand} < +2^{63}$ ); otherwise, set to 0.
C0, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FRNDINT—Round to Integer

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FC	FRNDINT	Valid	Valid	Round ST(0) to an integer.

### Description

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is  $\infty$ , the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$ST(0) \leftarrow \text{RoundToIntegralValue}(ST(0));$

### FPU Flags Affected

C1                      Set to 0 if stack underflow occurred.  
                             Set if result was rounded up; cleared otherwise.  
 C0, C2, C3            Undefined.

### Floating-Point Exceptions

#IS                      Stack underflow occurred.  
 #IA                      Source operand is an SNaN value or unsupported format.  
 #D                      Source operand is a denormal value.  
 #P                      Source operand is not an integral value.

### Protected Mode Exceptions

#NM                      CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
 #MF                      If there is a pending x87 FPU exception.  
 #UD                      If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FRSTOR—Restore x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DD /4	FRSTOR <i>m94/108byte</i>	Valid	Valid	Load FPU state from <i>m94byte</i> or <i>m108byte</i> .

### Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated. To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```

FPUControlWord ← SRC[FPUControlWord];
FPUStatusWord ← SRC[FPUStatusWord];
FPUTagWord ← SRC[FPUTagWord];
FPUDataPointer ← SRC[FPUDataPointer];
FPUInstructionPointer ← SRC[FPUInstructionPointer];
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode];

```

```

ST(0) ← SRC[ST(0)];
ST(1) ← SRC[ST(1)];
ST(2) ← SRC[ST(2)];
ST(3) ← SRC[ST(3)];
ST(4) ← SRC[ST(4)];
ST(5) ← SRC[ST(5)];
ST(6) ← SRC[ST(6)];
ST(7) ← SRC[ST(7)];

```

### FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

### Floating-Point Exceptions

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.



## FSAVE/FNSAVE—Store x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B DD /6	FSAVE <i>m94/108byte</i>	Valid	Valid	Store FPU state to <i>m94byte</i> or <i>m108byte</i> after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE* <i>m94/108byte</i>	Valid	Valid	Store FPU environment to <i>m94byte</i> or <i>m108byte</i> without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see “FINIT/FNINIT—Initialize Floating-Point Unit” in this chapter).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a “clean” FPU to a procedure.

The assembler issues two instructions for the FSAVE instruction (an FWAIT instruction followed by an FNSAVE instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

For Intel math coprocessors and FPU prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps ensure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

## Operation

(\* Save FPU State and Registers \*)

```
DEST[FPUControlWord] ← FPUControlWord;
DEST[FPUStatusWord] ← FPUStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUInstructionPointer] ← FPUInstructionPointer;
DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;
```

```
DEST[ST(0)] ← ST(0);
DEST[ST(1)] ← ST(1);
DEST[ST(2)] ← ST(2);
DEST[ST(3)] ← ST(3);
DEST[ST(4)] ← ST(4);
DEST[ST(5)] ← ST(5);
DEST[ST(6)] ← ST(6);
DEST[ST(7)] ← ST(7);
```

(\* Initialize FPU \*)

```
FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

## FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

#GP(0)	<p>If destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## FSCALE—Scale

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FD	FSCALE	Valid	Valid	Scale ST(0) by ST(1).

### Description

Truncates the value in the source operand (toward 0) to an integral value and adds that value to the exponent of the destination operand. The destination and source operands are floating-point values located in registers ST(0) and ST(1), respectively. This instruction provides rapid multiplication or division by integral powers of 2. The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-34. FSCALE Results**

ST(0)	ST(1)							
		$-\infty$	$-F$	$-0$	$+0$	$+F$	$+\infty$	NaN
	$-\infty$	NaN	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN
	$-F$	$-0$	$-F$	$-F$	$-F$	$-F$	$-\infty$	NaN
	$-0$	$-0$	$-0$	$-0$	$-0$	$-0$	NaN	NaN
	$+0$	$+0$	$+0$	$+0$	$+0$	$+0$	NaN	NaN
	$+F$	$+0$	$+F$	$+F$	$+F$	$+F$	$+\infty$	NaN
	$+\infty$	NaN	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

#### NOTES:

F Means finite floating-point value.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$$ST(0) \leftarrow ST(0) * 2^{\text{RoundTowardZero}(ST(1))},$$

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FSIN—Sine

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FE	FSIN	Valid	Valid	Replace ST(0) with the approximate of its sine.

### Description

Computes an approximation of the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range  $-2^{63}$  to  $+2^{63}$ . The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

**Table 3-35. FSIN Results**

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
$-F$	$-1$ to $+1$
$-0$	$-0$
$+0$	$+0$
$+F$	$-1$ to $+1$
$+\infty$	*
NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range  $-2^{63}$  to  $+2^{63}$  can be reduced to the range of the instruction by subtracting an appropriate integer multiple of  $2\pi$ . However, even within the range  $-2^{63}$  to  $+2^{63}$ , inaccurate results can occur because the finite approximation of  $\pi$  used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSIN only to arguments reduced accurately in software, to a value smaller in absolute value than  $3\pi/4$ . See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for  $\pi$  in performing such reductions.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```
IF  $-2^{63} < \text{ST}(0) < 2^{63}$ 
  THEN
    C2  $\leftarrow$  0;
    ST(0)  $\leftarrow$  fsin(ST(0)); // approximation of the mathematical sin function
  ELSE (* Source operand out of range *)
    C2  $\leftarrow$  1;
FI;
```

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred.  
 Set if result was rounded up; cleared otherwise.  
 C2 Set to 1 if outside range ( $-2^{63} < \text{source operand} < +2^{63}$ ); otherwise, set to 0.  
 C0, C3 Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FSINCOS—Sine and Cosine

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FB	FSINCOS	Valid	Valid	Compute the sine and cosine of ST(0); replace ST(0) with the approximate sine, and push the approximate cosine onto the register stack.

### Description

Computes both the approximate sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range  $-2^{63}$  to  $+2^{63}$ . The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

**Table 3-36. FSINCOS Results**

SRC	DEST	
ST(0)	ST(1) Cosine	ST(0) Sine
$-\infty$	*	*
$-F$	$-1$ to $+1$	$-1$ to $+1$
$-0$	$+1$	$-0$
$+0$	$+1$	$+0$
$+F$	$-1$ to $+1$	$-1$ to $+1$
$+\infty$	*	*
NaN	NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range  $-2^{63}$  to  $+2^{63}$  can be reduced to the range of the instruction by subtracting an appropriate integer multiple of  $2\pi$ . However, even within the range  $-2^{63}$  to  $+2^{63}$ , inaccurate results can occur because the finite approximation of  $\pi$  used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSINCOS only to arguments reduced accurately in software, to a value smaller in absolute value than  $3\pi/8$ . See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for  $\pi$  in performing such reductions.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.



**Operation**

```

IF ST(0) < 263
    THEN
        C2 ← 0;
        TEMP ← fcos(ST(0)); // approximation of cosine
        ST(0) ← fsin(ST(0)); // approximation of sine
        TOP ← TOP – 1;
        ST(0) ← TEMP;
    ELSE (* Source operand out of range *)
        C2 ← 1;
FI;

```

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurs. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range ( $-2^{63} < \text{source operand} < +2^{63}$ ); otherwise, set to 0.
C0, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FSQRT—Square Root

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 FA	FSQRT	Valid	Valid	Computes square root of ST(0) and stores the result in ST(0).

### Description

Computes the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-37. FSQRT Results**

SRC (ST(0))	DEST (ST(0))
$-\infty$	*
$-F$	*
$-0$	$-0$
$+0$	$+0$
$+F$	$+F$
$+\infty$	$+\infty$
NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

ST(0)  $\leftarrow$  SquareRoot(ST(0));

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format. Source operand is a negative value (except for $-0$ ).
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### **Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

### **Compatibility Mode Exceptions**

Same exceptions as in protected mode.

### **64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FST/FSTP—Store Floating Point Value

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 /2	FST <i>m32fp</i>	Valid	Valid	Copy ST(0) to <i>m32fp</i> .
DD /2	FST <i>m64fp</i>	Valid	Valid	Copy ST(0) to <i>m64fp</i> .
DD D0+i	FST ST(i)	Valid	Valid	Copy ST(0) to ST(i).
D9 /3	FSTP <i>m32fp</i>	Valid	Valid	Copy ST(0) to <i>m32fp</i> and pop register stack.
DD /3	FSTP <i>m64fp</i>	Valid	Valid	Copy ST(0) to <i>m64fp</i> and pop register stack.
DB /7	FSTP <i>m80fp</i>	Valid	Valid	Copy ST(0) to <i>m80fp</i> and pop register stack.
DD D8+i	FSTP ST(i)	Valid	Valid	Copy ST(0) to ST(i) and pop register stack.

### Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU register stack. When storing the value in memory, the value is converted to single-precision or double-precision floating-point format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also store values in memory in double extended-precision floating-point format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single-precision or double-precision, the significand of the value being stored is rounded to the width of the destination (according to the rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is  $\pm 0$ ,  $\pm\infty$ , or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value's identity as a 0,  $\infty$ , or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

DEST  $\leftarrow$  ST(0);

```
IF Instruction = FSTP
    THEN
        PopRegisterStack;
FI;
```

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred.  
Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0  $\leftarrow$  not roundup; 1  $\leftarrow$  roundup.

C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	If destination result is an SNaN value or unsupported format, except when the destination format is in double extended-precision floating-point format.
#U	Result is too small for the destination format.
#O	Result is too large for the destination format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FSTCW/FNSTCW—Store x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B D9 /7	FSTCW <i>m2byte</i>	Valid	Valid	Store FPU control word to <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW* <i>m2byte</i>	Valid	Valid	Store FPU control word to <i>m2byte</i> without checking for pending unmasked floating-point exceptions.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

The assembler issues two instructions for the FSTCW instruction (an FWAIT instruction followed by an FNSTCW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTCW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

### Operation

DEST ← FPUControlWord;

### FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FSTENV/FNSTENV—Store x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
9B D9 /6	FSTENV <i>m14/28byte</i>	Valid	Valid	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV <sup>*</sup> <i>m14/28byte</i>	Valid	Valid	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not. The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

The assembler issues two instructions for the FSTENV instruction (an FWAIT instruction followed by an FNSTENV instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTENV instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

### Operation

```
DEST[FPUControlWord] ← FPUControlWord;
DEST[FPUStatusWord] ← FPUStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUInstructionPointer] ← FPUInstructionPointer;
DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;
```

### FPU Flags Affected

The C0, C1, C2, and C3 are undefined.



## Floating-Point Exceptions

None

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FSTSW/FNSTSW—Store x87 FPU Status Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DD /7	FSTSW <i>m2byte</i>	Valid	Valid	Store FPU status word at <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
9B DF E0	FSTSW AX	Valid	Valid	Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.
DD /7	FNSTSW* <i>m2byte</i>	Valid	Valid	Store FPU status word at <i>m2byte</i> without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW* AX	Valid	Valid	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Description

Stores the current value of the x87 FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See the section titled “Branching and Conditional Moves on FPU Condition Codes” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.) This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

The assembler issues two instructions for the FSTSW instruction (an FWAIT instruction followed by an FNSTSW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTSW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled “No-Wait FPU Instructions Can Get FPU Interrupt in Window” in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of these circumstances. An FNSTSW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

### Operation

DEST ← FPUStatusWord;

### FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

### Floating-Point Exceptions

None

**Protected Mode Exceptions**

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D8 /4	FSUB <i>m32fp</i>	Valid	Valid	Subtract <i>m32fp</i> from ST(0) and store result in ST(0).
DC /4	FSUB <i>m64fp</i>	Valid	Valid	Subtract <i>m64fp</i> from ST(0) and store result in ST(0).
D8 E0+i	FSUB ST(0), ST(i)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(0).
DC E8+i	FSUB ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(i).
DE E8+i	FSUBP ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i), store result in ST(i), and pop register stack.
DE E9	FSUBP	Valid	Valid	Subtract ST(0) from ST(1), store result in ST(1), and pop register stack.
DA /4	FISUB <i>m32int</i>	Valid	Valid	Subtract <i>m32int</i> from ST(0) and store result in ST(0).
DE /4	FISUB <i>m16int</i>	Valid	Valid	Subtract <i>m16int</i> from ST(0) and store result in ST(0).

### Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a floating-point or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

Table 3-38 shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST – SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward  $-\infty$  mode, in which case the result is  $-0$ . This instruction also guarantees that  $+0 - (-0) = +0$ , and that  $-0 - (+0) = -0$ . When the source operand is an integer 0, it is treated as a +0.

When one operand is  $\infty$ , the result is  $\infty$  of the expected sign. If both operands are  $\infty$  of the same sign, an invalid-operation exception is generated.

Table 3-38. FSUB/FSUBP/FISUB Results

DEST	SRC							
		$-\infty$	$-F$ or $-I$	$-0$	$+0$	$+F$ or $+I$	$+\infty$	NaN
	$-\infty$	*	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN
	$-F$	$+\infty$	$\pm F$ or $\pm 0$	DEST	DEST	$-F$	$-\infty$	NaN
	$-0$	$+\infty$	$-\text{SRC}$	$\pm 0$	$-0$	$-\text{SRC}$	$-\infty$	NaN
	$+0$	$+\infty$	$-\text{SRC}$	$+0$	$\pm 0$	$-\text{SRC}$	$-\infty$	NaN
	$+F$	$+\infty$	$+F$	DEST	DEST	$\pm F$ or $\pm 0$	$-\infty$	NaN
	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

F Means finite floating-point value.

I Means integer.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FISUB

THEN

DEST  $\leftarrow$  DEST – ConvertToDoubleExtendedPrecisionFP(SRC);

ELSE (\* Source operand is floating-point value \*)

DEST  $\leftarrow$  DEST – SRC;

FI;

IF Instruction = FSUBP

THEN

PopRegisterStack;

FI;

**FPU Flags Affected**

- C1 Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.
- C0, C2, C3 Undefined.

**Floating-Point Exceptions**

- #IS Stack underflow occurred.
- #IA Operand is an SNaN value or unsupported format.  
Operands are infinities of like sign.
- #D Source operand is a denormal value.
- #U Result is too small for destination format.
- #O Result is too large for destination format.
- #P Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FSUBR/FSUBRP/FISUBR—Reverse Subtract

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /5	FSUBR <i>m32fp</i>	Valid	Valid	Subtract ST(0) from <i>m32fp</i> and store result in ST(0).
DC /5	FSUBR <i>m64fp</i>	Valid	Valid	Subtract ST(0) from <i>m64fp</i> and store result in ST(0).
D8 E8+i	FSUBR ST(0), ST(i)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(0).
DC E0+i	FSUBR ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(i).
DE E0+i	FSUBRP ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0), store result in ST(i), and pop register stack.
DE E1	FSUBRP	Valid	Valid	Subtract ST(1) from ST(0), store result in ST(1), and pop register stack.
DA /5	FISUBR <i>m32int</i>	Valid	Valid	Subtract ST(0) from <i>m32int</i> and store result in ST(0).
DE /5	FISUBR <i>m16int</i>	Valid	Valid	Subtract ST(0) from <i>m16int</i> and store result in ST(0).

### Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a floating-point or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value (SRC – DEST = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward  $-\infty$  mode, in which case the result is  $-0$ . This instruction also guarantees that  $+0 - (-0) = +0$ , and that  $-0 - (+0) = -0$ . When the source operand is an integer 0, it is treated as a +0.

When one operand is  $\infty$ , the result is  $\infty$  of the expected sign. If both operands are  $\infty$  of the same sign, an invalid-operation exception is generated.

**Table 3-39. FSUBR/FSUBRP/FISUBR Results**

DEST	SRC							
		$-\infty$	$-F$ or $-I$	$-0$	$+0$	$+F$ or $+I$	$+\infty$	NaN
	$-\infty$	*	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN
	$-F$	$-\infty$	$\pm F$ or $\pm 0$	$-\text{DEST}$	$-\text{DEST}$	$+F$	$+\infty$	NaN
	$-0$	$-\infty$	SRC	$\pm 0$	$+0$	SRC	$+\infty$	NaN
	$+0$	$-\infty$	SRC	$-0$	$\pm 0$	SRC	$+\infty$	NaN
	$+F$	$-\infty$	$-F$	$-\text{DEST}$	$-\text{DEST}$	$\pm F$ or $\pm 0$	$+\infty$	NaN
	$+\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

F Means finite floating-point value.

I Means integer.

\* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

IF Instruction = FISUBR

THEN

DEST  $\leftarrow$  ConvertToDoubleExtendedPrecisionFP(SRC)  $-$  DEST;

ELSE (\* Source operand is floating-point value \*)

DEST  $\leftarrow$  SRC  $-$  DEST; FI;

IF Instruction = FSUBRP

THEN

PopRegisterStack; FI;

**FPU Flags Affected**

C1 Set to 0 if stack underflow occurred.  
Set if result was rounded up; cleared otherwise.

C0, C2, C3 Undefined.

**Floating-Point Exceptions**

#IS Stack underflow occurred.

#IA Operand is an SNaN value or unsupported format.  
Operands are infinities of like sign.

#D Source operand is a denormal value.

#U Result is too small for destination format.

#O Result is too large for destination format.

#P Value cannot be represented exactly in destination format.



**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## FTST—TEST

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E4	FTST	Valid	Valid	Compare ST(0) with 0.0.

### Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

**Table 3-40. FTST Results**

Condition	C3	C2	C0
ST(0) > 0.0	0	0	0
ST(0) < 0.0	0	0	1
ST(0) = 0.0	1	0	0
Unordered	1	1	1

This instruction performs an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to “unordered” and the invalid operation exception is generated.

The sign of zero is ignored, so that  $(-0.0 \leftarrow +0.0)$ .

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

CASE (relation of operands) OF

Not comparable: C3, C2, C0  $\leftarrow$  111;  
 ST(0) > 0.0: C3, C2, C0  $\leftarrow$  000;  
 ST(0) < 0.0: C3, C2, C0  $\leftarrow$  001;  
 ST(0) = 0.0: C3, C2, C0  $\leftarrow$  100;

ESAC;

### FPU Flags Affected

C1 Set to 0.  
 C0, C2, C3 See Table 3-40.

### Floating-Point Exceptions

#IS Stack underflow occurred.  
 #IA The source operand is a NaN value or is in an unsupported format.  
 #D The source operand is a denormal value.

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
 #MF If there is a pending x87 FPU exception.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### **Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

### **Compatibility Mode Exceptions**

Same exceptions as in protected mode.

### **64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
DD E0+i	FUCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
DD E1	FUCOM	Valid	Valid	Compare ST(0) with ST(1).
DD E8+i	FUCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
DD E9	FUCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DA E9	FUCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

### Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that  $-0.0$  is equal to  $+0.0$ .

**Table 3-41. FUCOM/FUCOMP/FUCOMPP Results**

Comparison Results*	C3	C2	C0
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered	1	1	1

#### NOTES:

\* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine Floating-Point” in this chapter). The FUCOM/FUCOMP/FUCOMPP instructions perform the same operations as the FCOM/FCOMP/FCOMPP instructions. The only difference is that the FUCOM/FUCOMP/FUCOMPP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM/FCOMP/FCOMPP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM/FCOMP/FCOMPP instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

## Operation

CASE (relation of operands) OF

ST > SRC: C3, C2, C0 ← 000;

ST < SRC: C3, C2, C0 ← 001;

ST = SRC: C3, C2, C0 ← 100;

ESAC;

IF ST(0) or SRC = QNaN, but not SNaN or unsupported format

THEN

C3, C2, C0 ← 111;

ELSE (\* ST(0) or SRC is SNaN or unsupported format \*)

#IA;

IF FPUControlWord.IM = 1

THEN

C3, C2, C0 ← 111;

FI;

FI;

IF Instruction = FUCOMP

THEN

PopRegisterStack;

FI;

IF Instruction = FUCOMPP

THEN

PopRegisterStack;

FI;

## FPU Flags Affected

C1 Set to 0 if stack underflow occurred.

C0, C2, C3 See Table 3-41.

## Floating-Point Exceptions

#IS Stack underflow occurred.

#IA One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.

#D One or both operands are denormal values.

## Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FXAM—Examine Floating-Point

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E5	FXAM	Valid	Valid	Classify value or number in ST(0).

### Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

**Table 3-42. FXAM Results**

Class	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal finite number	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal number	1	1	0

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$C1 \leftarrow \text{sign bit of ST}; (* 0 \text{ for positive, } 1 \text{ for negative } *)$

CASE (class of value or number in ST(0)) OF

  Unsupported: C3, C2, C0  $\leftarrow$  000;

  NaN: C3, C2, C0  $\leftarrow$  001;

  Normal: C3, C2, C0  $\leftarrow$  010;

  Infinity: C3, C2, C0  $\leftarrow$  011;

  Zero: C3, C2, C0  $\leftarrow$  100;

  Empty: C3, C2, C0  $\leftarrow$  101;

  Denormal: C3, C2, C0  $\leftarrow$  110;

ESAC;

### FPU Flags Affected

C1 Sign of value in ST(0).

C0, C2, C3 See Table 3-42.

### Floating-Point Exceptions

None

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.



## FXCH—Exchange Register Contents

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 C8+i	FXCH ST(i)	Valid	Valid	Exchange the contents of ST(0) and ST(i).
D9 C9	FXCH	Valid	Valid	Exchange the contents of ST(0) and ST(1).

### Description

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

IF (Number-of-operands) is 1

THEN

temp  $\leftarrow$  ST(0);

ST(0)  $\leftarrow$  SRC;

SRC  $\leftarrow$  temp;

ELSE

temp  $\leftarrow$  ST(0);

ST(0)  $\leftarrow$  ST(1);

ST(1)  $\leftarrow$  temp;

FI;

### FPU Flags Affected

C1 Set to 0.

C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow occurred.

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

#MF If there is a pending x87 FPU exception.

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

**FXRSTOR—Restore x87 FPU, MMX, XMM, and MXCSR State**

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF AE /1 FXRSTOR <i>m512byte</i>	M	Valid	Valid	Restore the x87 FPU, MMX, XMM, and MXCSR register state from <i>m512byte</i> .
NP REX.W + OF AE /1 FXRSTOR64 <i>m512byte</i>	M	Valid	N.E.	Restore the x87 FPU, MMX, XMM, and MXCSR register state from <i>m512byte</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

**Description**

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layouts of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode FXSAVE/FXRSTOR with REX.W=0, and the third format is for 64-bit mode with FXSAVE64/FXRSTOR64. Table 3-43 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-46 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-47 shows the layout of the 64-bit mode state information when REX.W is clear (FXSAVE/FXRSTOR).

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Table 3-43, Table 3-46, or Table 3-47. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

If the OSFXSR bit in control register CR4 is not set, the FXRSTOR instruction may not restore the states of the XMM and MXCSR registers. This behavior is implementation dependent.

If the MXCSR state contains an unmasked exception with a corresponding status flag also set, loading the register with the FXRSTOR instruction will not result in a SIMD floating-point error condition being generated. Only the next occurrence of this unmasked exception will result in the exception being generated.

Bits 16 through 32 of the MXCSR register are defined as reserved and should be set to 0. Attempting to write a 1 in any of these bits from the saved state image will result in a general protection exception (#GP) being generated.

Bytes 464:511 of an FXSAVE image are available for software use. FXRSTOR ignores the content of bytes 464:511 in an FXSAVE state image.

**Operation**

IF 64-Bit Mode

THEN

(x87 FPU, MMX, XMM15-XMM0, MXCSR) Load(SRC);

ELSE

(x87 FPU, MMX, XMM7-XMM0, MXCSR) ← Load(SRC);

FI;

**x87 FPU and SIMD Floating-Point Exceptions**

None.

## Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See alignment check exception [#AC] below.) For an attempt to set reserved bits in MXCSR.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If instruction is preceded by a LOCK prefix.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH. For an attempt to set reserved bits in MXCSR.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment. For an attempt to set reserved bits in MXCSR.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If instruction is preceded by a LOCK prefix.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## FXSAVE—Save x87 FPU, MMX Technology, and SSE State

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF AE /0 FXSAVE <i>m512byte</i>	M	Valid	Valid	Save the x87 FPU, MMX, XMM, and MXCSR register state to <i>m512byte</i> .
NP REX.W + OF AE /0 FXSAVE64 <i>m512byte</i>	M	Valid	N.E.	Save the x87 FPU, MMX, XMM, and MXCSR register state to <i>m512byte</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512 byte region depends on whether the processor is operating in non-64-bit operating modes or 64-bit sub-mode of IA-32e mode.

Bytes 464:511 are available to software use. The processor does not write to bytes 464:511 of an FXSAVE area.

The operation of FXSAVE in non-64-bit modes is described first.

### Non-64-Bit Mode Operation

Table 3-43 shows the layout of the state information in memory when the processor is operating in legacy modes.

**Table 3-43. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR  
Memory Region**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Rsvd		FCS		FIP[31:0]				FOP		Rsvd	FTW	FSW		FCW		0
MXCSR_MASK				MXCSR				Rsrvd		FDS		FDP[31:0]				16
Reserved						ST0/MM0										32
Reserved						ST1/MM1										48
Reserved						ST2/MM2										64
Reserved						ST3/MM3										80
Reserved						ST4/MM4										96
Reserved						ST5/MM5										112
Reserved						ST6/MM6										128
Reserved						ST7/MM7										144
XMM0																160
XMM1																176
XMM2																192
XMM3																208
XMM4																224
XMM5																240
XMM6																256
XMM7																272
Reserved																288

**Table 3-43. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR  
Memory Region (Contd.)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved																304
Reserved																320
Reserved																336
Reserved																352
Reserved																368
Reserved																384
Reserved																400
Reserved																416
Reserved																432
Reserved																448
Available																464
Available																480
Available																496

The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-43 are defined in Table 3-44.

**Table 3-44. Field Definitions**

Field	Definition
FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i> , for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i> , for the layout of the x87 FPU status word.
Abridged FTW	x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs.
FOP	x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i> , for the layout of the x87 FPU opcode field.
FIP	x87 FPU Instruction Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit IP offset. 16-bit mode — low 16 bits are IP offset; high 16 bits are reserved. 64-bit mode with REX.W — 64-bit IP offset. 64-bit mode without REX.W — 32-bit IP offset. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1</i> , for a description of the x87 FPU instruction pointer.

**Table 3-44. Field Definitions (Contd.)**

Field	Definition
FCS	x87 FPU Instruction Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.
FDP	x87 FPU Instruction Operand (Data) Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit DP offset. 16-bit mode — low 16 bits are DP offset; high 16 bits are reserved. 64-bit mode with REX.W — 64-bit DP offset. 64-bit mode without REX.W — 32-bit DP offset. See “x87 FPU Instruction and Operand (Data) Pointers” in Chapter 8 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for a description of the x87 FPU operand pointer.
FDS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for the layout of the MXCSR register. If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save this register. This behavior is implementation dependent.
MXCSR_MASK	MXCSR_MASK (32 bits). This mask can be used to adjust values written to the MXCSR register, ensuring that reserved bits are set to 0. Set the mask bits and flags in MXCSR to the mode of operation desired for SSE and SSE2 SIMD floating-point instructions. See “Guidelines for Writing to the MXCSR Register” in Chapter 11 of the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1</i> , for instructions for how to determine and use the MXCSR_MASK value.
ST0/MM0 through ST7/MM7	x87 FPU or MMX technology registers. These 80-bit fields contain the x87 FPU data registers or the MMX technology registers, depending on the state of the processor prior to the execution of the FXSAVE instruction. If the processor had been executing x87 FPU instruction prior to the FXSAVE instruction, the x87 FPU data registers are saved; if it had been executing MMX instructions (or SSE or SSE2 instructions that operated on the MMX technology registers), the MMX technology registers are saved. When the MMX technology registers are saved, the high 16 bits of the field are reserved.
XMM0 through XMM7	XMM registers (128 bits per field). If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save these registers. This behavior is implementation dependent.

The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field (unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

```

R7 R6 R5 R4 R3 R2 R1 R0
11 xx xx xx 11 11 11 11

```

Here, 11B indicates empty stack elements and “xx” indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

```

R7 R6 R5 R4 R3 R2 R1 R0
0 1 1 1 0 0 0 0

```

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, the as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be



used by an application program to pass a “clean” x87 FPU state to a procedure, since it retains the current state. To clean the x87 FPU state, an application must explicitly execute an FINIT instruction after an FXSAVE instruction to reinitialize the x87 FPU state.

- The format of the memory image saved with the FXSAVE instruction is the same regardless of the current addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management). This behavior differs from the FSAVE instructions, where the memory image format is different depending on the addressing mode and operating mode. Because of the different image formats, the memory image saved with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the stored data was not the contents of MMX technology registers) using Table 3-45.

**Table 3-45. Recreating FSAVE Format**

Exponent all 1's	Exponent all 0's	Fraction all 0's	J and M bits	FTW valid bit	x87 FTW	
0	0	0	0x	1	Special	10
0	0	0	1x	1	Valid	00
0	0	1	00	1	Special	10
0	0	1	10	1	Valid	00
0	1	0	0x	1	Special	10
0	1	0	1x	1	Special	10
0	1	1	00	1	Zero	01
0	1	1	10	1	Special	10
1	0	0	1x	1	Special	10
1	0	0	1x	1	Special	10
1	0	1	00	1	Special	10
1	0	1	10	1	Special	10
For all legal combinations above.				0	Empty	11

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0's.

### IA-32e Mode Operation

In compatibility sub-mode of IA-32e mode, legacy SSE registers, XMM0 through XMM7, are saved according to the legacy FXSAVE map. In 64-bit mode, all of the SSE registers, XMM0 through XMM15, are saved. Additionally, there are two different layouts of the FXSAVE map in 64-bit mode, corresponding to FXSAVE64 (which requires REX.W=1) and FXSAVE (REX.W=0). In the FXSAVE64 map (Table 3-46), the FPU IP and FPU DP pointers are 64-bit wide. In the FXSAVE map for 64-bit mode (Table 3-47), the FPU IP and FPU DP pointers are 32-bits.

**Table 3-46. Layout of the 64-bit-mode FXSAVE64 Map  
(requires REX.W = 1)**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
FIP								FOP		Reserved	FTW	FSW		FCW		0
MXCSR_MASK				MXCSR				FDP								16
Reserved						ST0/MM0										32
Reserved						ST1/MM1										48
Reserved						ST2/MM2										64
Reserved						ST3/MM3										80
Reserved						ST4/MM4										96
Reserved						ST5/MM5										112
Reserved						ST6/MM6										128
Reserved						ST7/MM7										144
								XMM0								160
								XMM1								176
								XMM2								192
								XMM3								208
								XMM4								224
								XMM5								240
								XMM6								256
								XMM7								272
								XMM8								288
								XMM9								304
								XMM10								320
								XMM11								336
								XMM12								352
								XMM13								368
								XMM14								384
								XMM15								400
								Reserved								416
								Reserved								432
								Reserved								448
								Available								464
								Available								480
								Available								496

**Table 3-47. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0)**

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Reserved	FCS	FIP[31:0]		FOP	Reserved	FTW	FSW	FCW	0
MXCSR_MASK		MXCSR		Reserved	FDS		FDP[31:0]		16
Reserved			ST0/MM0						32
Reserved			ST1/MM1						48
Reserved			ST2/MM2						64
Reserved			ST3/MM3						80
Reserved			ST4/MM4						96
Reserved			ST5/MM5						112
Reserved			ST6/MM6						128
Reserved			ST7/MM7						144
XMM0									160
XMM1									176
XMM2									192
XMM3									208
XMM4									224
XMM5									240
XMM6									256
XMM7									272
XMM8									288
XMM9									304
XMM10									320
XMM11									336
XMM12									352
XMM13									368
XMM14									384
XMM15									400
Reserved									416
Reserved									432
Reserved									448
Available									464
Available									480
Available									496

## Operation

```

IF 64-Bit Mode
    THEN
        IF REX.W = 1
            THEN
                DEST ← Save64BitPromotedFxsave(x87 FPU, MMX, XMM15-XMM0,
                    MXCSR);
            ELSE
                DEST ← Save64BitDefaultFxsave(x87 FPU, MMX, XMM15-XMM0, MXCSR);
        FI;
    ELSE
        DEST ← SaveLegacyFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);
FI;

```

## Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If a memory operand is not aligned on a 16-byte boundary, regardless of segment. (See the description of the alignment check exception [#AC] below.)
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.
#UD	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

**Implementation Note**

The order in which the processor signals general-protection (#GP) and page-fault (#PF) exceptions when they both occur on an instruction boundary is given in Table 5-2 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This order vary for FXSAVE for different processor implementations.

## FXTRACT—Extract Exponent and Significand

Opcode/ Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F4 FXTRACT	Valid	Valid	Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.

### Description

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a floating-point value. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand's true (unbiased) exponent expressed as a floating-point value. (The operation performed by this instruction is a superset of the IEEE-recommended  $\log_b(x)$  function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in double extended-precision floating-point format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of  $-\infty$  is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```
TEMP ← Significand(ST(0));
ST(0) ← Exponent(ST(0));
TOP ← TOP - 1;
ST(0) ← TEMP;
```

### FPU Flags Affected

C1 Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.  
C0, C2, C3 Undefined.

### Floating-Point Exceptions

#IS Stack underflow or overflow occurred.  
#IA Source operand is an SNaN value or unsupported format.  
#Z ST(0) operand is  $\pm 0$ .  
#D Source operand is a denormal value.

### Protected Mode Exceptions

#NM CR0.EM[bit 2] or CR0.TS[bit 3] = 1.  
#MF If there is a pending x87 FPU exception.  
#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### **Compatibility Mode Exceptions**

Same exceptions as in protected mode.

### **64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## FYL2X—Compute $y * \log_2 x$

Opcode	Instruction	64-Bit Mode	Compat/Leg Mode	Description
D9 F1	FYL2X	Valid	Valid	Replace ST(1) with $(ST(1) * \log_2 ST(0))$ and pop the register stack.

### Description

Computes  $(ST(1) * \log_2 (ST(0)))$ , stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

**Table 3-48. FYL2X Results**

		ST(0)							
ST(1)		$-\infty$	$-F$	$\pm 0$	$+0 < +F < +1$	$+1$	$+F > +1$	$+\infty$	NaN
	$-\infty$	*	*	$+\infty$	$+\infty$	*	$-\infty$	$-\infty$	NaN
	$-F$	*	*	**	$+F$	$-0$	$-F$	$-\infty$	NaN
	$-0$	*	*	*	$+0$	$-0$	$-0$	*	NaN
	$+0$	*	*	*	$-0$	$+0$	$+0$	*	NaN
	$+F$	*	*	**	$-F$	$+0$	$+F$	$+\infty$	NaN
	$+\infty$	*	*	$-\infty$	$-\infty$	*	$+\infty$	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-operation (#IA) exception.

\*\* Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains  $\pm 0$ , the instruction returns  $\infty$  with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base (b):

$$\log_b x \leftarrow (\log_2 b)^{-1} * \log_2 x$$

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$ST(1) \leftarrow ST(1) * \log_2 ST(0);$

PopRegisterStack;

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.



### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN or unsupported format. Source operand in register ST(0) is a negative finite value (not -0).
#Z	Source operand in register ST(0) is $\pm 0$ .
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## FYL2XP1—Compute $y * \log_2(x + 1)$

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F9	FYL2XP1	Valid	Valid	Replace ST(1) with $ST(1) * \log_2(ST(0) + 1.0)$ and pop the register stack.

### Description

Computes  $(ST(1) * \log_2(ST(0) + 1.0))$ , stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

$$-(1 - \sqrt{2}/2) \text{ to } (1 - \sqrt{2}/2)$$

The source operand in ST(1) can range from  $-\infty$  to  $+\infty$ . If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

**Table 3-49. FYL2XP1 Results**

		ST(0)				
ST(1)		$-(1 - (\sqrt{2}/2)) \text{ to } -0$	-0	+0	+0 to $+(1 - (\sqrt{2}/2))$	NaN
	$-\infty$	$+\infty$	*	*	$-\infty$	NaN
	-F	+F	+0	-0	-F	NaN
	-0	+0	+0	-0	-0	NaN
	+0	-0	-0	+0	+0	NaN
	+F	-F	-0	+0	+F	NaN
	$+\infty$	$-\infty$	*	*	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN

#### NOTES:

F Means finite floating-point value.

\* Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. For small epsilon ( $\epsilon$ ) values, more significant digits can be retained by using the FYL2XP1 instruction than by using  $(\epsilon+1)$  as an argument to the FYL2X instruction. The  $(\epsilon+1)$  expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where  $n$  is the logarithm base desired for the result of the FYL2XP1 instruction:

$$\text{scale factor} \leftarrow \log_n 2$$

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

$ST(1) \leftarrow ST(1) * \log_2(ST(0) + 1.0);$

PopRegisterStack;

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## HADDPD—Packed Double-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7C /r HADDPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Horizontal add packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 7C /r VHADDPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Horizontal add packed double-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 7C /r VHADDPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Horizontal add packed double-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

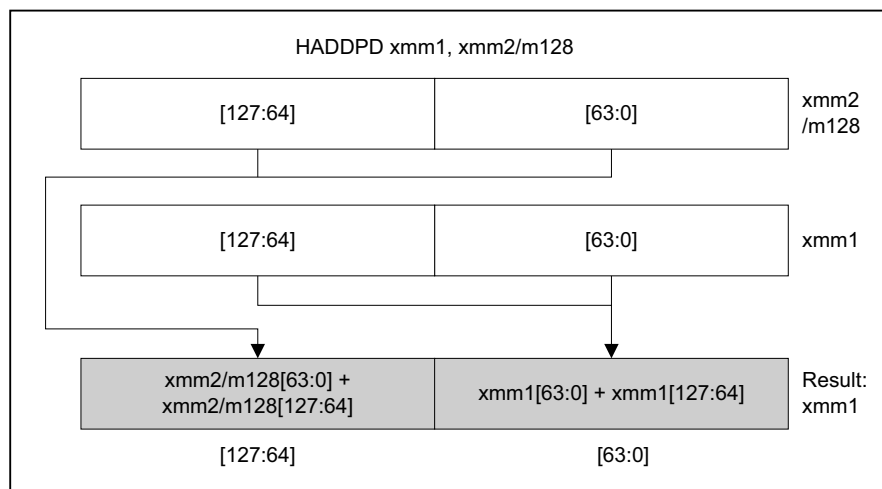
### Description

Adds the double-precision floating-point values in the high and low quadwords of the destination operand and stores the result in the low quadword of the destination operand.

Adds the double-precision floating-point values in the high and low quadwords of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-16 for HADDPD; see Figure 3-17 for VHADDPD.



OM15993

Figure 3-16. HADDPD—Packed Double-FP Horizontal Add

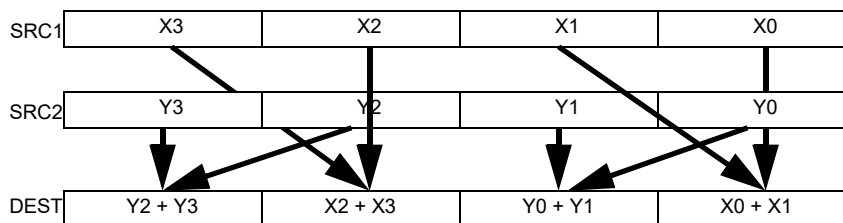


Figure 3-17. VHADDPD operation

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### HADDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

#### VHADDPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[MAXVL-1:128] ← 0
```

#### VHADDPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[191:128] ← SRC1[255:192] + SRC1[191:128]
DEST[255:192] ← SRC2[255:192] + SRC2[191:128]
```

### Intel C/C++ Compiler Intrinsic Equivalent

VHADDPD: `__m256d _mm256_hadd_pd (__m256d a, __m256d b);`

HADDPD: `__m128d _mm_hadd_pd (__m128d a, __m128d b);`

### Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

**Numeric Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2.

**HADDPS—Packed Single-FP Horizontal Add**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7C /r HADDPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Horizontal add packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG 7C /r VHADDPS <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Horizontal add packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.F2.0F.WIG 7C /r VHADDPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Horizontal add packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Adds the single-precision floating-point values in the first and second dwords of the destination operand and stores the result in the first dword of the destination operand.

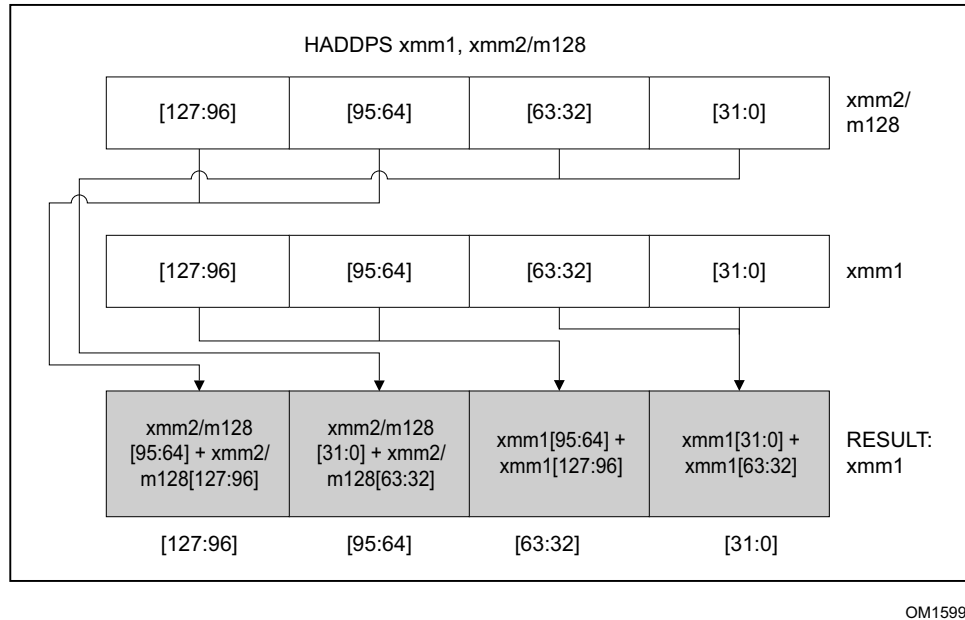
Adds single-precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

Adds single-precision floating-point values in the first and second dword of the source operand and stores the result in the third dword of the destination operand.

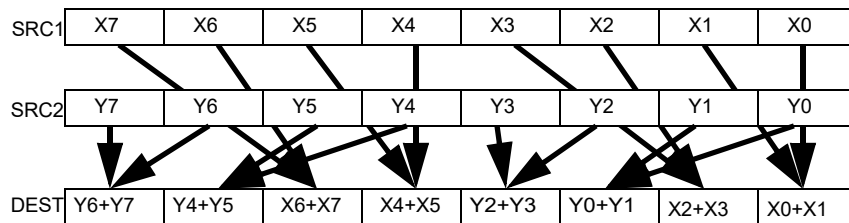
Adds single-precision floating-point values in the third and fourth dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-18 for HADDPS; see Figure 3-19 for VHADDPS.



**Figure 3-18. HADDPS—Packed Single-FP Horizontal Add**



**Figure 3-19. VHADDPS operation**

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.



## Operation

### HADDPS (128-bit Legacy SSE version)

$DEST[31:0] \leftarrow SRC1[63:32] + SRC1[31:0]$   
 $DEST[63:32] \leftarrow SRC1[127:96] + SRC1[95:64]$   
 $DEST[95:64] \leftarrow SRC2[63:32] + SRC2[31:0]$   
 $DEST[127:96] \leftarrow SRC2[127:96] + SRC2[95:64]$   
 $DEST[MAXVL-1:128]$  (Unmodified)

### VHADDPS (VEX.128 encoded version)

$DEST[31:0] \leftarrow SRC1[63:32] + SRC1[31:0]$   
 $DEST[63:32] \leftarrow SRC1[127:96] + SRC1[95:64]$   
 $DEST[95:64] \leftarrow SRC2[63:32] + SRC2[31:0]$   
 $DEST[127:96] \leftarrow SRC2[127:96] + SRC2[95:64]$   
 $DEST[MAXVL-1:128] \leftarrow 0$

### VHADDPS (VEX.256 encoded version)

$DEST[31:0] \leftarrow SRC1[63:32] + SRC1[31:0]$   
 $DEST[63:32] \leftarrow SRC1[127:96] + SRC1[95:64]$   
 $DEST[95:64] \leftarrow SRC2[63:32] + SRC2[31:0]$   
 $DEST[127:96] \leftarrow SRC2[127:96] + SRC2[95:64]$   
 $DEST[159:128] \leftarrow SRC1[191:160] + SRC1[159:128]$   
 $DEST[191:160] \leftarrow SRC1[255:224] + SRC1[223:192]$   
 $DEST[223:192] \leftarrow SRC2[191:160] + SRC2[159:128]$   
 $DEST[255:224] \leftarrow SRC2[255:224] + SRC2[223:192]$

## Intel C/C++ Compiler Intrinsic Equivalent

HADDPS: `__m128 _mm_hadd_ps (__m128 a, __m128 b);`

VHADDPS: `__m256 _mm256_hadd_ps (__m256 a, __m256 b);`

## Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

## Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

## Other Exceptions

See Exceptions Type 2.

## HLT—Halt

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F4	HLT	Z0	Valid	Valid	Halt

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

Enter Halt state;

### Flags Affected

None

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

HSUBPD—Packed Double-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7D /r HSUBPD <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Horizontal subtract packed double-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG 7D /r VHSUBPD <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Horizontal subtract packed double-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.66.0F.WIG 7D /r VHSUBPD <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Horizontal subtract packed double-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

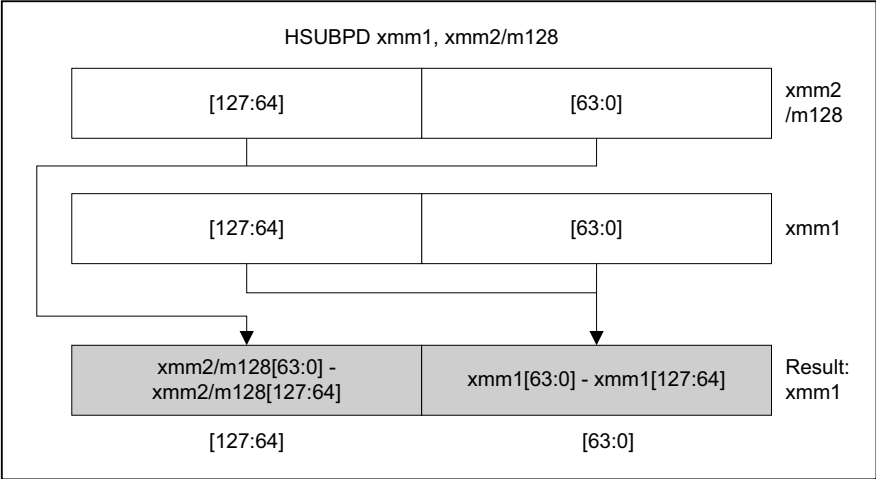
The HSUBPD instruction subtracts horizontally the packed DP FP numbers of both operands.

Subtracts the double-precision floating-point value in the high quadword of the destination operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand.

Subtracts the double-precision floating-point value in the high quadword of the source operand from the low quadword of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-20 for HSUBPD; see Figure 3-21 for VHSUBPD.



OM15995

Figure 3-20. HSUBPD—Packed Double-FP Horizontal Subtract

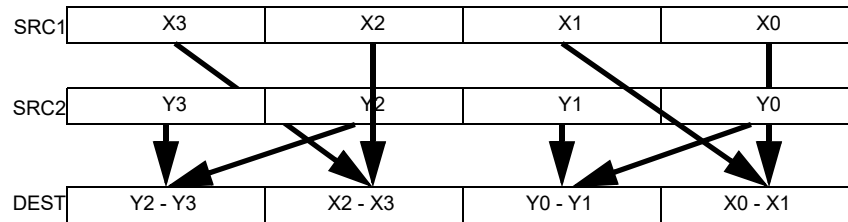


Figure 3-21. VHSUBPD operation

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### HSUBPD (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

#### VHSUBPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[MAXVL-1:128] ← 0
```

#### VHSUBPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[191:128] ← SRC1[191:128] - SRC1[255:192]
DEST[255:192] ← SRC2[191:128] - SRC2[255:192]
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
HSUBPD:   __m128d _mm_hsub_pd(__m128d a, __m128d b)
VHSUBPD:  __m256d _mm256_hsub_pd (__m256d a, __m256d b);
```

### Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

### Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

See Exceptions Type 2.

## HSUBPS—Packed Single-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7D /r HSUBPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE3	Horizontal subtract packed single-precision floating-point values from <i>xmm2/m128</i> to <i>xmm1</i> .
VEX.NDS.128.F2.0F.WIG 7D /r VHSUBPS <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Horizontal subtract packed single-precision floating-point values from <i>xmm2</i> and <i>xmm3/mem</i> .
VEX.NDS.256.F2.0F.WIG 7D /r VHSUBPS <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX	Horizontal subtract packed single-precision floating-point values from <i>ymm2</i> and <i>ymm3/mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Subtracts the single-precision floating-point value in the second dword of the destination operand from the first dword of the destination operand and stores the result in the first dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

Subtracts the single-precision floating-point value in the second dword of the source operand from the first dword of the source operand and stores the result in the third dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-22 for HSUBPS; see Figure 3-23 for VHSUBPS.

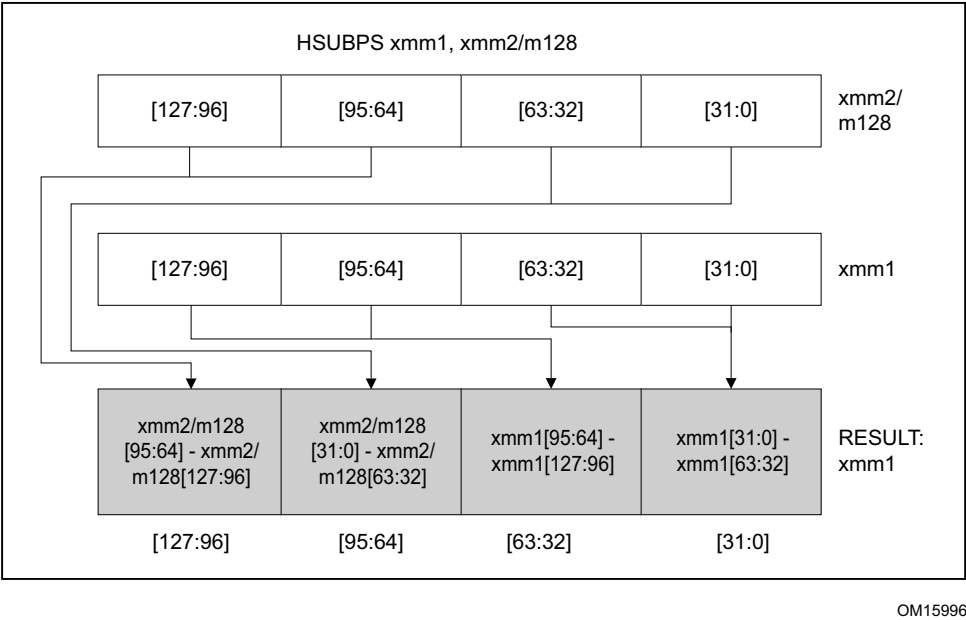


Figure 3-22. HSUBPS—Packed Single-FP Horizontal Subtract

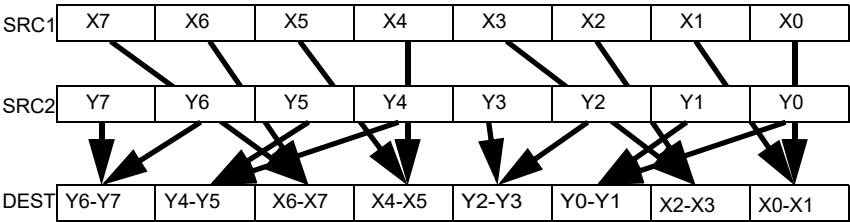


Figure 3-23. VHSUBPS operation

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

## Operation

### HSUBPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

### VHSUBPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$

### VHSUBPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC1}[63:32]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[95:64] - \text{SRC1}[127:96]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC2}[31:0] - \text{SRC2}[63:32]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC2}[95:64] - \text{SRC2}[127:96]$   
 $\text{DEST}[159:128] \leftarrow \text{SRC1}[159:128] - \text{SRC1}[191:160]$   
 $\text{DEST}[191:160] \leftarrow \text{SRC1}[223:192] - \text{SRC1}[255:224]$   
 $\text{DEST}[223:192] \leftarrow \text{SRC2}[159:128] - \text{SRC2}[191:160]$   
 $\text{DEST}[255:224] \leftarrow \text{SRC2}[223:192] - \text{SRC2}[255:224]$

## Intel C/C++ Compiler Intrinsic Equivalent

HSUBPS: `__m128 _mm_hsub_ps(__m128 a, __m128 b);`

VHSUBPS: `__m256 _mm256_hsub_ps (__m256 a, __m256 b);`

## Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

## Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

## Other Exceptions

See Exceptions Type 2.



## IDIV—Signed Divide

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /7	IDIV <i>r/m8</i>	M	Valid	Valid	Signed divide AX by <i>r/m8</i> , with result stored in: AL ← Quotient, AH ← Remainder.
REX + F6 /7	IDIV <i>r/m8</i> *	M	Valid	N.E.	Signed divide AX by <i>r/m8</i> , with result stored in AL ← Quotient, AH ← Remainder.
F7 /7	IDIV <i>r/m16</i>	M	Valid	Valid	Signed divide DX:AX by <i>r/m16</i> , with result stored in AX ← Quotient, DX ← Remainder.
F7 /7	IDIV <i>r/m32</i>	M	Valid	Valid	Signed divide EDX:EAX by <i>r/m32</i> , with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /7	IDIV <i>r/m64</i>	M	Valid	N.E.	Signed divide RDX:RAX by <i>r/m64</i> , with result stored in RAX ← Quotient, RDX ← Remainder.

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m ( <i>r</i> )	NA	NA	NA

### Description

Divides the (signed) value in the AX, DX:AX, or EDX:EAX (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor).

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the signed value in RDX:RAX by the source operand. RAX contains a 64-bit quotient; RDX contains a 64-bit remainder.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-50.

**Table 3-50. IDIV Results**

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	<i>r/m8</i>	AL	AH	–128 to +127
Doubleword/word	DX:AX	<i>r/m16</i>	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	<i>r/m32</i>	EAX	EDX	–2 <sup>31</sup> to 2 <sup>31</sup> – 1
Doublequadword/quadword	RDX:RAX	<i>r/m64</i>	RAX	RDX	–2 <sup>63</sup> to 2 <sup>63</sup> – 1

## Operation

```

IF SRC = 0
    THEN #DE; (* Divide error *)
FI;

IF OperandSize = 8 (* Word/byte operation *)
    THEN
        temp ← AX / SRC; (* Signed division *)
        IF (temp > 7FH) or (temp < 80H)
            (* If a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* Divide error *)
            ELSE
                AL ← temp;
                AH ← AX SignedModulus SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp ← DX:AX / SRC; (* Signed division *)
            IF (temp > 7FFFH) or (temp < 8000H)
                (* If a positive result is greater than 7FFFH
                or a negative result is less than 8000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    AX ← temp;
                    DX ← DX:AX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
        THEN
            temp ← EDX:EAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFH) or (temp < 80000000H)
                (* If a positive result is greater than 7FFFFFFFFFH
                or a negative result is less than 80000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    EAX ← temp;
                    EDX ← EDX:EAX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp ← RDX:RAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
                (* If a positive result is greater than 7FFFFFFFFFFFFFFFFFH
                or a negative result is less than 8000000000000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    RAX ← temp;
                    RDX ← RDX:RAX SignedModulus SRC;
            FI;
        FI;
    FI;
FI;

```

**Flags Affected**

The CF, OF, SF, ZF, AF, and PF flags are undefined.

**Protected Mode Exceptions**

#DE	If the source operand (divisor) is 0. The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#DE	If the source operand (divisor) is 0.  The signed result (quotient) is too large for the destination.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#DE	If the source operand (divisor) is 0.  The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#DE	If the source operand (divisor) is 0 If the quotient is too large for the designated register.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## IMUL—Signed Multiply

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /5	IMUL <i>r/m8</i> *	M	Valid	Valid	AX ← AL * <i>r/m</i> byte.
F7 /5	IMUL <i>r/m16</i>	M	Valid	Valid	DX:AX ← AX * <i>r/m</i> word.
F7 /5	IMUL <i>r/m32</i>	M	Valid	Valid	EDX:EAX ← EAX * <i>r/m32</i> .
REX.W + F7 /5	IMUL <i>r/m64</i>	M	Valid	N.E.	RDX:RAX ← RAX * <i>r/m64</i> .
0F AF /r	IMUL <i>r16, r/m16</i>	RM	Valid	Valid	word register ← word register * <i>r/m16</i> .
0F AF /r	IMUL <i>r32, r/m32</i>	RM	Valid	Valid	doubleword register ← doubleword register * <i>r/m32</i> .
REX.W + 0F AF /r	IMUL <i>r64, r/m64</i>	RM	Valid	N.E.	Quadword register ← Quadword register * <i>r/m64</i> .
6B /r ib	IMUL <i>r16, r/m16, imm8</i>	RMI	Valid	Valid	word register ← <i>r/m16</i> * sign-extended immediate byte.
6B /r ib	IMUL <i>r32, r/m32, imm8</i>	RMI	Valid	Valid	doubleword register ← <i>r/m32</i> * sign-extended immediate byte.
REX.W + 6B /r ib	IMUL <i>r64, r/m64, imm8</i>	RMI	Valid	N.E.	Quadword register ← <i>r/m64</i> * sign-extended immediate byte.
69 /r iw	IMUL <i>r16, r/m16, imm16</i>	RMI	Valid	Valid	word register ← <i>r/m16</i> * immediate word.
69 /r id	IMUL <i>r32, r/m32, imm32</i>	RMI	Valid	Valid	doubleword register ← <i>r/m32</i> * immediate doubleword.
REX.W + 69 /r id	IMUL <i>r64, r/m64, imm32</i>	RMI	Valid	N.E.	Quadword register ← <i>r/m64</i> * immediate doubleword.

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8/16/32	NA

### Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product (twice the size of the input operand) is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.
- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.
- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The intermediate product (twice the size of the first source operand) is truncated and stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- **One-operand form** —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.
- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

## Operation

```

IF (NumberOfOperands = 1)
  THEN IF (OperandSize = 8)
    THEN
      TMP_XP ← AL * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
      AX ← TMP_XP[15:0];
      IF SignExtend(TMP_XP[7:0]) = TMP_XP
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1; FI;
    ELSE IF OperandSize = 16
      THEN
        TMP_XP ← AX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        DX:AX ← TMP_XP[31:0];
        IF SignExtend(TMP_XP[15:0]) = TMP_XP
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; FI;
    ELSE IF OperandSize = 32
      THEN
        TMP_XP ← EAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        EDX:EAX ← TMP_XP[63:0];
        IF SignExtend(TMP_XP[31:0]) = TMP_XP
          THEN CF ← 0; OF ← 0;
          ELSE CF ← 1; OF ← 1; FI;
    ELSE (* OperandSize = 64 *)
      TMP_XP ← RAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
      EDX:EAX ← TMP_XP[127:0];
      IF SignExtend(TMP_XP[63:0]) = TMP_XP
        THEN CF ← 0; OF ← 0;
        ELSE CF ← 1; OF ← 1; FI;
    FI;
  FI;

```

```

    FI;
ELSE IF (NumberOfOperands = 2)
    THEN
        TMP_XP ← DEST * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
        DEST ← TruncateToOperandSize(TMP_XP);
        IF SignExtend(DEST) ≠ TMP_XP
            THEN CF ← 1; OF ← 1;
            ELSE CF ← 0; OF ← 0; FI;
    ELSE (* NumberOfOperands = 3 *)
        TMP_XP ← SRC1 * SRC2 (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC1 *)
        DEST ← TruncateToOperandSize(TMP_XP);
        IF SignExtend(DEST) ≠ TMP_XP
            THEN CF ← 1; OF ← 1;
            ELSE CF ← 0; OF ← 0; FI;
    FI;
FI;

```

### Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## IN—Input from Port

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
E4 <i>ib</i>	IN AL, <i>imm8</i>	I	Valid	Valid	Input byte from <i>imm8</i> I/O port address into AL.
E5 <i>ib</i>	IN AX, <i>imm8</i>	I	Valid	Valid	Input word from <i>imm8</i> I/O port address into AX.
E5 <i>ib</i>	IN EAX, <i>imm8</i>	I	Valid	Valid	Input dword from <i>imm8</i> I/O port address into EAX.
EC	IN AL,DX	ZO	Valid	Valid	Input byte from I/O port in DX into AL.
ED	IN AX,DX	ZO	Valid	Valid	Input word from I/O port in DX into AX.
ED	IN EAX,DX	ZO	Valid	Valid	Input doubleword from I/O port in DX into EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	<i>imm8</i>	NA	NA	NA
ZO	NA	NA	NA	NA

### Description

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size. At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

```

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST ← SRC; (* Read from selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Read from selected I/O port *)
    FI;

```

### Flags Affected

None



### Protected Mode Exceptions

- |        |  |
|--------|--|
| #GP(0) | If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. |
| #UD    | If the LOCK prefix is used.  |

### Real-Address Mode Exceptions

- |     |                             |
|-----|-----------------------------|
| #UD | If the LOCK prefix is used. |
|-----|-----------------------------|

### Virtual-8086 Mode Exceptions

- |                 |  |
|-----------------|--|
| #GP(0)          | If any of the I/O permission bits in the TSS for the I/O port being accessed is 1. |
| #PF(fault-code) | If a page fault occurs.  |
| #UD             | If the LOCK prefix is used.  |

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- |        |  |
|--------|--|
| #GP(0) | If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. |
| #UD    | If the LOCK prefix is used.  |

## INC—Increment by 1

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FE /0	INC <i>r/m8</i>	M	Valid	Valid	Increment <i>r/m</i> byte by 1.
REX + FE /0	INC <i>r/m8</i> *	M	Valid	N.E.	Increment <i>r/m</i> byte by 1.
FF /0	INC <i>r/m16</i>	M	Valid	Valid	Increment <i>r/m</i> word by 1.
FF /0	INC <i>r/m32</i>	M	Valid	Valid	Increment <i>r/m</i> doubleword by 1.
REX.W + FF /0	INC <i>r/m64</i>	M	Valid	N.E.	Increment <i>r/m</i> quadword by 1.
40+ <i>rw</i> **	INC <i>r16</i>	O	N.E.	Valid	Increment word register by 1.
40+ <i>rd</i>	INC <i>r32</i>	O	N.E.	Valid	Increment doubleword register by 1.

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

\*\* 40H through 47H are REX prefixes in 64-bit mode.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m ( <i>r</i> , <i>w</i> )	NA	NA	NA
O	opcode + <i>rd</i> ( <i>r</i> , <i>w</i> )	NA	NA	NA

### Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, INC *r16* and INC *r32* are not encodable (because opcodes 40H through 47H are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

### Operation

DEST ← DEST + 1;

### AFlags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

### Protected Mode Exceptions

#GP(0)	If the destination operand is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
6C	INS <i>m8</i> , DX	Z0	Valid	Valid	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.*
6D	INS <i>m16</i> , DX	Z0	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. <sup>1</sup>
6D	INS <i>m32</i> , DX	Z0	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. <sup>1</sup>
6C	INSB	Z0	Valid	Valid	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI. <sup>1</sup>
6D	INSW	Z0	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. <sup>1</sup>
6D	INSD	Z0	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI. <sup>1</sup>

### NOTES:

\* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:DI, ES:EDI or the RDI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The ES segment cannot be overridden with a segment override prefix.) The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the INS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand must be “DX,” and the destination operand should be a symbol that indicates the size of the I/O port and the destination address. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the INS instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the INS instructions. Here also DX is assumed by the processor to be the source operand and ES:(E)DI is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: INSB (byte), INSW (word), or INSD (doubleword).

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the DI/EDI/RDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 18, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, default address size is 64 bits, 32 bit address size is supported using the prefix 67H. The address of the memory destination is specified by RDI or EDI. 16-bit address size is not supported in 64-bit mode. The operand size is not promoted.

These instructions may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g. #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

## Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST ← SRC; (* Read from I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL IOPL *)
        DEST ← SRC; (* Read from I/O port *)
    FI;
```

Non-64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E)DI ← (E)DI + 1;
        ELSE (E)DI ← (E)DI - 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI ← (E)DI + 2;
            ELSE (E)DI ← (E)DI - 2; FI;
        ELSE (* Doubleword transfer *)
            THEN IF DF = 0
                THEN (E)DI ← (E)DI + 4;
                ELSE (E)DI ← (E)DI - 4; FI;
        FI;
    FI;
```

FI64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E|R)DI ← (E|R)DI + 1;
        ELSE (E|R)DI ← (E|R)DI - 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI ← (E)DI + 2;
            ELSE (E)DI ← (E)DI - 2; FI;
        ELSE (* Doubleword transfer *)
```

```

        THEN IF DF = 0
            THEN (E|R)DI ← (E|R)DI + 4;
            ELSE (E|R)DI ← (E|R)DI - 4; FI;
    FI;
FI;

```

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)	<p>If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.</p> <p>If the destination is located in a non-writable segment.</p> <p>If an illegal memory operand effective address in the ES segments is given.</p>
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	<p>If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.</p> <p>If the memory address is in a non-canonical form.</p>
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## INSERTPS—Insert Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS xmm1, xmm2/m32, imm8	A	V/V	SSE4_1	Insert a single-precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.
VEX.NDS.128.66.0F3A.WIG 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.
EVEX.NDS.128.66.0F3A.W0 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	C	V/V	AVX512F	Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

(register source form)

Copy a single-precision scalar floating-point element into a 128-bit vector register. The immediate operand has three fields, where the ZMask bits specify which elements of the destination will be set to zero, the Count\_D bits specify which element of the destination will be overwritten with the scalar value, and for vector register sources the Count\_S bits specify which element of the source will be copied. When the scalar source is a memory operand the Count\_S bits are ignored.

(memory source form)

Load a floating-point element from a 32-bit memory location and destination operand it into the first source at the location indicated by the Count\_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.128 and EVEX encoded version: The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

If VINSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

## Operation

### VINSERTPS (VEX.128 and EVEX encoded version)

```

IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
    ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
    0: TMP ← SRC2[31:0]
    1: TMP ← SRC2[63:32]
    2: TMP ← SRC2[95:64]
    3: TMP ← SRC2[127:96]
ESAC;
CASE (COUNT_D) OF
    0: TMP2[31:0] ← TMP
        TMP2[127:32] ← SRC1[127:32]
    1: TMP2[63:32] ← TMP
        TMP2[31:0] ← SRC1[31:0]
        TMP2[127:64] ← SRC1[127:64]
    2: TMP2[95:64] ← TMP
        TMP2[63:0] ← SRC1[63:0]
        TMP2[127:96] ← SRC1[127:96]
    3: TMP2[127:96] ← TMP
        TMP2[95:0] ← SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
    ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H
    ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
    ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H
    ELSE DEST[127:96] ← TMP2[127:96]
DEST[MAXVL-1:128] ← 0

```

### INSERTPS (128-bit Legacy SSE version)

```

IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
    ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
    0: TMP ← SRC[31:0]
    1: TMP ← SRC[63:32]
    2: TMP ← SRC[95:64]
    3: TMP ← SRC[127:96]
ESAC;

CASE (COUNT_D) OF
    0: TMP2[31:0] ← TMP
        TMP2[127:32] ← DEST[127:32]
    1: TMP2[63:32] ← TMP
        TMP2[31:0] ← DEST[31:0]
        TMP2[127:64] ← DEST[127:64]
    2: TMP2[95:64] ← TMP

```



```

    TMP2[63:0] ← DEST[63:0]
    TMP2[127:96] ← DEST[127:96]
    3: TMP2[127:96] ← TMP
    TMP2[95:0] ← DEST[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
    ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H
    ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
    ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H
    ELSE DEST[127:96] ← TMP2[127:96]
DEST[MAXVL-1:128] (Unmodified)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VINSETPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
INSETRTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E9NF.

## INT *n*/INTO/INT 3—Call to Interrupt Procedure

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CC	INT 3	Z0	Valid	Valid	Interrupt 3—trap to debugger.
CD <i>ib</i>	INT <i>imm8</i>	I	Valid	Valid	Interrupt vector specified by immediate byte.
CE	INTO	Z0	Invalid	Valid	Interrupt 4—if overflow flag is 1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA
I	imm8	NA	NA	NA

### Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled “Interrupts and Exceptions” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code).

An interrupt generated by INTO or INT3 (CC) differs from one generated by INT *n* in the following ways:

- The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
- The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.

(These features do not pertain to CD03, the “normal” 2-byte opcode for INT 3. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.)

The action of the INT *n* instruction (including the INTO and INT 3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT *n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and its pointers are called **interrupt vectors**.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the “Operation” section for this instruction (except #GP).

Table 3-51. Decision Table

PE	0	1	1	1	1	1	1	1
VM	-	-	-	-	-	0	1	1
IOPL	-	-	-	-	-	-	<3	=3
DPL/CPL RELATIONSHIP	-	DPL < CPL	-	DPL > CPL	DPL = CPL or C	DPL < CPL & NC	-	-
INTERRUPT TYPE	-	S/W	-	-	-	-	-	-
GATE TYPE	-	-	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

**NOTES:**

- Don't Care.
- Y Yes, action taken.
- Blank Action not taken.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT *n* instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

**Operation**

The following operational description applies not only to the INT *n* and INTO instructions, but also to external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.

The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction `error_code(num,idt,ext)`, where `idt` and `ext` are bit values. The pseudofunction produces an error code as follows: (1) if `idt` is 0, the error code is `(num & FCH) | ext`; (2) if `idt` is 1, the error code is `(num << 3) | 2 | ext`.

In many cases, the pseudofunction `error_code` is invoked with a pseudovisible EXT. The value of EXT depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt, EXT is 0; otherwise, EXT is 1.

```

IF PE = 0
  THEN
    GOTO REAL-ADDRESS-MODE;
  ELSE (* PE = 1 *)
    IF (EFLAGS.VM = 1 AND CR4.VME = 0 AND IOPL < 3 AND INT n)
      THEN
        #GP(0); (* Bit 0 of error code is 0 because INT n *)
      ELSE
        IF (EFLAGS.VM = 1 AND CR4.VME = 1 AND INT n)
          THEN
            Consult bit n of the software interrupt redirection bit map in the TSS;
            IF bit n is clear
              THEN (* redirect interrupt to 8086 program interrupt handler *)
                Push EFLAGS[15:0]; (* if IOPL < 3, save VIF in IF position and save IOPL position as 3 *)
                Push CS;
                Push IP;
                IF IOPL = 3
                  THEN IF ← 0; (* Clear interrupt flag *)
                  ELSE VIF ← 0; (* Clear virtual interrupt flag *)
                FI;
                TF ← 0; (* Clear trap flag *)
                load CS and EIP (lower 16 bits only) from entry n in interrupt vector table referenced from TSS;
              ELSE
                IF IOPL = 3
                  THEN GOTO PROTECTED-MODE;
                  ELSE #GP(0); (* Bit 0 of error code is 0 because INT n *)
                FI;
            FI;
          ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
            IF (IA32_EFER.LMA = 0)
              THEN (* Protected mode, or virtual-8086 mode interrupt *)
                GOTO PROTECTED-MODE;
              ELSE (* IA-32e mode interrupt *)
                GOTO IA-32e-MODE;
            FI;
          FI;
        FI;
      FI;
    FI;
  REAL-ADDRESS-MODE:
    IF ((vector_number << 2) + 3) is not within IDT limit
      THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
      THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ← 0; (* Clear interrupt flag *)
    TF ← 0; (* Clear trap flag *)
    AC ← 0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed in real-address mode*)
    CS ← IDT(Descriptor (vector_number << 2), selector);
    EIP ← IDT(Descriptor (vector_number << 2), offset); (* 16 bit offset AND 0000FFFFH *)
  END;
  PROTECTED-MODE:

```

```

IF ((vector_number << 3) + 7) is not within IDT limits
or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
    THEN #GP(error_code(vector_number,1,EXT)); FI;
    (* idt operand to error_code set because vector is used *)
IF software interrupt (* Generated by INT n, INT3, or INTO *)
    THEN
        IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
            THEN #GP(error_code(vector_number,1,0)); FI;
            (* idt operand to error_code set because vector is used *)
            (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
        FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF task gate (* Specified in the selected interrupt table descriptor *)
        THEN GOTO TASK-GATE;
        ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
    FI;
END;
IA-32e-MODE:
IF INTO and CS.L = 1 (64-bit mode)
    THEN #UD;
    FI;
IF ((vector_number << 4) + 15) is not in IDT limits
or selected IDT descriptor is not an interrupt-, or trap-gate type
    THEN #GP(error_code(vector_number,1,EXT));
    (* idt operand to error_code set because vector is used *)
    FI;
IF software interrupt (* Generated by INT n, INT 3, or INTO *)
    THEN
        IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
            THEN #GP(error_code(vector_number,1,0));
            (* idt operand to error_code set because vector is used *)
            (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
        FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
    FI;
    GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read TSS selector in task gate (IDT descriptor);
    IF local/global bit is set to local or index not within GDT limits
        THEN #GP(error_code(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Access TSS descriptor in GDT;
    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(TSS selector,0,EXT); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF TSS not present
        THEN #NP(TSS selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)

```

```

SWITCH-TASKS (with nesting) to TSS;
IF interrupt caused by fault with error code
    THEN
        IF stack limit does not allow push of error code
            THEN #SS(EXT); FI;
        Push(error code);
    FI;
IF EIP not within code segment limit
    THEN #GP(EXT); FI;
END;
TRAP-OR-INTERRUPT-GATE:
    Read new code-segment selector for trap or interrupt gate (IDT descriptor);
    IF new code-segment selector is NULL
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read descriptor referenced by new code-segment selector;
    IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code-segment descriptor is not present,
        THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code segment is non-conforming with DPL < CPL
        THEN
            IF VM = 0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
                    DPL < CPL *)
                ELSE (* VM = 1 *)
                    IF new code-segment DPL ≠ 0
                        THEN #GP(error_code(new code-segment selector,0,EXT));
                        (* idt operand to error_code is 0 because selector is used *)
                    GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
                    (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
                FI;
            ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
                IF VM = 1
                    THEN #GP(error_code(new code-segment selector,0,EXT));
                    (* idt operand to error_code is 0 because selector is used *)
                IF new code segment is conforming or new code-segment DPL = CPL
                    THEN
                        GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                    ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
                        #GP(error_code(new code-segment selector,0,EXT));
                        (* idt operand to error_code is 0 because selector is used *)
                    FI;
                FI;
            FI;
        END;
    INTER-PRIVILEGE-LEVEL-INTERRUPT:
        (* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
        IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)

```

THEN

(\* Identify stack-segment selector for new privilege level in current TSS \*)

IF current TSS is 32-bit

THEN

TSSstackAddress  $\leftarrow$  (new code-segment DPL  $\ll$  3) + 4;

IF (TSSstackAddress + 5) > current TSS limit

THEN #TS(error\_code(current TSS selector,0,EXT)); FI;

(\* idt operand to error\_code is 0 because selector is used \*)

NewSS  $\leftarrow$  2 bytes loaded from (TSS base + TSSstackAddress + 4);

NewESP  $\leftarrow$  4 bytes loaded from (TSS base + TSSstackAddress);

ELSE (\* current TSS is 16-bit \*)

TSSstackAddress  $\leftarrow$  (new code-segment DPL  $\ll$  2) + 2

IF (TSSstackAddress + 3) > current TSS limit

THEN #TS(error\_code(current TSS selector,0,EXT)); FI;

(\* idt operand to error\_code is 0 because selector is used \*)

NewSS  $\leftarrow$  2 bytes loaded from (TSS base + TSSstackAddress + 2);

NewESP  $\leftarrow$  2 bytes loaded from (TSS base + TSSstackAddress);

FI;

IF NewSS is NULL

THEN #TS(EXT); FI;

IF NewSS index is not within its descriptor-table limits

or NewSS RPL  $\neq$  new code-segment DPL

THEN #TS(error\_code(NewSS,0,EXT)); FI;

(\* idt operand to error\_code is 0 because selector is used \*)

Read new stack-segment descriptor for NewSS in GDT or LDT;

IF new stack-segment DPL  $\neq$  new code-segment DPL

or new stack-segment Type does not indicate writable data segment

THEN #TS(error\_code(NewSS,0,EXT)); FI;

(\* idt operand to error\_code is 0 because selector is used \*)

IF NewSS is not present

THEN #SS(error\_code(NewSS,0,EXT)); FI;

(\* idt operand to error\_code is 0 because selector is used \*)

ELSE (\* IA-32e mode \*)

IF IDT-gate IST = 0

THEN TSSstackAddress  $\leftarrow$  (new code-segment DPL  $\ll$  3) + 4;

ELSE TSSstackAddress  $\leftarrow$  (IDT gate IST  $\ll$  3) + 28;

FI;

IF (TSSstackAddress + 7) > current TSS limit

THEN #TS(error\_code(current TSS selector,0,EXT)); FI;

(\* idt operand to error\_code is 0 because selector is used \*)

NewRSP  $\leftarrow$  8 bytes loaded from (current TSS base + TSSstackAddress);

NewSS  $\leftarrow$  new code-segment DPL; (\* NULL selector with RPL = new CPL \*)

FI;

IF IDT gate is 32-bit

THEN

IF new stack does not have room for 24 bytes (error code pushed)

or 20 bytes (no error code pushed)

THEN #SS(error\_code(NewSS,0,EXT)); FI;

(\* idt operand to error\_code is 0 because selector is used \*)

FI

ELSE

IF IDT gate is 16-bit

THEN

IF new stack does not have room for 12 bytes (error code pushed)

```

        or 10 bytes (no error code pushed);
        THEN #SS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    ELSE (* 64-bit IDT gate*)
        IF StackAddress is non-canonical
            THEN #SS(EXT); FI; (* Error code contains NULL selector *)
    FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        IF instruction pointer from IDT gate is not within new code-segment limits
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        ESP ← NewESP;
        SS ← NewSS; (* Segment descriptor information also loaded *)
    ELSE (* IA-32e mode *)
        IF instruction pointer from IDT gate contains a non-canonical address
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        RSP ← NewRSP & FFFFFFFF0H;
        SS ← NewSS;
    FI;
IF IDT gate is 32-bit
    THEN
        CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
    ELSE
        IF IDT gate 16-bit
            THEN
                CS:IP ← Gate(CS:IP);
                (* Segment descriptor information also loaded *)
            ELSE (* 64-bit IDT gate *)
                CS:RIP ← Gate(CS:RIP);
                (* Segment descriptor information also loaded *)
            FI;
    FI;
FI;
IF IDT gate is 32-bit
    THEN
        Push(far pointer to old stack);
        (* Old SS and ESP, 3 words padded to 4 *)
        Push(EFLAGS);
        Push(far pointer to return instruction);
        (* Old CS and EIP, 3 words padded to 4 *)
        Push(ErrorCode); (* If needed, 4 bytes *)
    ELSE
        IF IDT gate 16-bit
            THEN
                Push(far pointer to old stack);
                (* Old SS and SP, 2 words *)
                Push(EFLAGS(15-0));
                Push(far pointer to return instruction);
                (* Old CS and IP, 2 words *)
                Push(ErrorCode); (* If needed, 2 bytes *)
            ELSE (* 64-bit IDT gate *)
                Push(far pointer to old stack);
                (* Old SS and SP, each an 8-byte push *)
                Push(RFLAGS); (* 8-byte push *)
            FI;
    FI;

```



```

        Push(far pointer to return instruction);
        (* Old CS and RIP, each an 8-byte push *)
        Push(ErrorCode); (* If needed, 8-bytes *)

    FI;

FI;
CPL ← new code-segment DPL;
CS(RPL) ← CPL;
IF IDT gate is interrupt gate
    THEN IF ← 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
TF ← 0;
VM ← 0;
RF ← 0;
NT ← 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
(* Identify stack-segment selector for privilege level 0 in current TSS *)
IF current TSS is 32-bit
    THEN
        IF TSS limit < 9
            THEN #TS(error_code(current TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (current TSS base + 8);
            NewESP ← 4 bytes loaded from (current TSS base + 4);
        ELSE (* current TSS is 16-bit *)
            IF TSS limit < 5
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
                NewSS ← 2 bytes loaded from (current TSS base + 4);
                NewESP ← 2 bytes loaded from (current TSS base + 2);
            FI;
        IF NewSS is NULL
            THEN #TS(EXT); FI; (* Error code contains NULL selector *)
        IF NewSS index is not within its descriptor table limits
        or NewSS RPL ≠ 0
            THEN #TS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        Read new stack-segment descriptor for NewSS in GDT or LDT;
        IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
            THEN #TS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        IF new stack segment not present
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        IF IDT gate is 32-bit
            THEN
                IF new stack does not have room for 40 bytes (error code pushed)
                or 36 bytes (no error code pushed)
                    THEN #SS(error_code(NewSS,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                ELSE (* IDT gate is 16-bit *)
                    IF new stack does not have room for 20 bytes (error code pushed)
                    or 18 bytes (no error code pushed)
                        THEN #SS(error_code(NewSS,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)

```

```

FI;
IF instruction pointer from IDT gate is not within new code-segment limits
    THEN #GP(EXT); FI; (* Error code contains NULL selector *)
tempEFLAGS ← EFLAGS;
VM ← 0;
TF ← 0;
RF ← 0;
NT ← 0;
IF service through interrupt gate
    THEN IF = 0; FI;
TempSS ← SS;
TempESP ← ESP;
SS ← NewSS;
ESP ← NewESP;
(* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);
Push(TempEFlags);
Push(CS);
Push(EIP);
GS ← 0; (* Segment registers made NULL, invalid for use in protected mode *)
FS ← 0;
DS ← 0;
ES ← 0;
CS ← Gate(CS); (* Segment descriptor information also loaded *)
CS(RPL) ← 0;
CPL ← 0;
IF IDT gate is 32-bit
    THEN
        EIP ← Gate(instruction pointer);
    ELSE (* IDT gate is 16-bit *)
        EIP ← Gate(instruction pointer) AND 0000FFFFH;
FI;
(* Start execution of new routine in Protected Mode *)
END;
INTRA-PRIVILEGE-LEVEL-INTERRUPT:
(* PE = 1, DPL = CPL or conforming segment *)
IF IA32_EFER.LMA = 1 (* IA-32e mode *)
    IF IDT-descriptor IST ≠ 0
        THEN
            TSSstackAddress ← (IDT-descriptor IST « 3) + 28;
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
            ELSE NewRSP ← RSP;
        FI;
    FI;
    IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)

```

```

THEN
    IF current stack does not have room for 16 bytes (error code pushed)
    or 12 bytes (no error code pushed)
        THEN #SS(EXT); FI; (* Error code contains NULL selector *)
    ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
        IF current stack does not have room for 8 bytes (error code pushed)
        or 6 bytes (no error code pushed)
            THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
            IF NewRSP contains a non-canonical address
                THEN #SS(EXT); (* Error code contains NULL selector *)
        FI;
    FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        IF instruction pointer from IDT gate is not within new code-segment limit
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        ELSE
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP ← NewRSP & FFFFFFFF0H;
        FI;
    FI;
IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
    THEN
        Push (EFLAGS);
        Push (far pointer to return instruction); (* 3 words padded to 4 *)
        CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
        Push (ErrorCode); (* If any *)
    ELSE
        IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
            THEN
                Push (FLAGS);
                Push (far pointer to return location); (* 2 words *)
                CS:IP ← Gate(CS:IP);
                (* Segment descriptor information also loaded *)
                Push (ErrorCode); (* If any *)
            ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                Push(far pointer to old stack);
                (* Old SS and SP, each an 8-byte push *)
                Push(RFLAGS); (* 8-byte push *)
                Push(far pointer to return instruction);
                (* Old CS and RIP, each an 8-byte push *)
                Push(ErrorCode); (* If needed, 8 bytes *)
                CS:RIP ← GATE(CS:RIP);
                (* Segment descriptor information also loaded *)
            FI;
        FI;
    FI;
FI;
CS(RPL) ← CPL;
IF IDT gate is interrupt gate
    THEN IF ← 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
TF ← 0;
NT ← 0;
VM ← 0;
RF ← 0;

```

END;

### Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task's TSS.

### Protected Mode Exceptions

#GP(error_code)	<p>If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.</p> <p>If the segment selector in the interrupt-, trap-, or task gate is NULL.</p> <p>If an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.</p> <p>If an interrupt is generated by the INT <i>n</i>, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.</p> <p>If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(error_code)	<p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.</p> <p>If the SS register is being loaded and the segment pointed to is marked not present.</p> <p>If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.</p>
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	<p>If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.</p> <p>If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.</p> <p>If the stack segment selector in the TSS is NULL.</p> <p>If the stack segment for the TSS is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

### Real-Address Mode Exceptions

#GP	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the interrupt vector number is outside the IDT limits.</p>
#SS	<p>If stack limit violation on push.</p> <p>If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.</p>
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(error_code)	<p>(For INT <i>n</i>, INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3.</p> <p>If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.</p> <p>If the segment selector in the interrupt-, trap-, or task gate is NULL.</p> <p>If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.</p> <p>If an interrupt is generated by the INT <i>n</i> instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.</p> <p>If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p>
#SS(error_code)	<p>If the SS register is being loaded and the segment pointed to is marked not present.</p> <p>If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.</p>
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	<p>If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.</p> <p>If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.</p> <p>If the stack segment selector in the TSS is NULL.</p> <p>If the stack segment for the TSS is not a writable data segment.</p> <p>If segment-selector index for stack segment is outside descriptor table limits.</p>
#PF(fault-code)	If a page fault occurs.
#BP	If the INT 3 instruction is executed.
#OF	If the INTO instruction is executed and the OF flag is set.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(error_code)	<p>If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical.</p> <p>If the segment selector in the 64-bit interrupt or trap gate is NULL.</p> <p>If the vector selects a descriptor outside the IDT limits.</p> <p>If the vector points to a gate which is in non-canonical space.</p> <p>If the vector points to a descriptor which is not a 64-bit interrupt gate or 64-bit trap gate.</p> <p>If the descriptor pointed to by the gate selector is outside the descriptor table limit.</p> <p>If the descriptor pointed to by the gate selector is in non-canonical space.</p> <p>If the descriptor pointed to by the gate selector is not a code segment.</p> <p>If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set.</p> <p>If the descriptor pointed to by the gate selector has DPL &gt; CPL.</p>
#SS(error_code)	<p>If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.</p> <p>If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).</p>
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.
#TS(error_code)	<p>If an attempt to load RSP from the TSS causes an access to non-canonical space.</p> <p>If the RSP from the TSS is outside descriptor table limits.</p>
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

## INVD—Invalidate Internal Caches

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 08	INVD	Z0	Valid	Valid	Flush internal caches; initiate flushing of external caches.

### NOTES:

\* See the IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Invalidates (flushes) the processor's internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

The INVD instruction may be used when the cache is used as temporary memory and the cache contents need to be invalidated rather than written back to memory. When the cache is used as temporary memory, no external device should be actively writing data to main memory.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Note that any data from an external device to main memory (for example, via a PCIWrite) can be temporarily stored in the caches; these data can be lost when an INVD instruction is executed. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, temporary memory, testing, or fault recovery where cache coherency with main memory is not a concern), software should instead use the WBINVD instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

The INVD instruction is implementation dependent; it may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

### Operation

```
Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution *)
```

### Flags Affected

None

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)                    The INVD instruction cannot be executed in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.



## INVLPG—Invalidate TLB Entries

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01/7	INVLPG <i>m</i>	M	Valid	Valid	Invalidate TLB entries for page containing <i>m</i> .

### NOTES:

\* See the IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.<sup>1</sup>

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidate only TLB entries associated with the current PCID. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see “MOV—Move to/from Control Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B* and Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

This instruction’s operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

### IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

### Operation

Invalidate(RelevantTLBEntries);  
Continue; (\* Continue execution \*)

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.  
#UD Operand is a register.  
If the LOCK prefix is used.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*), the instruction invalidates all of them.

**Real-Address Mode Exceptions**

#UD                      Operand is a register.  
                            If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)                 The INVLPG instruction cannot be executed at the virtual-8086 mode.

**64-Bit Mode Exceptions**

#GP(0)                 If the current privilege level is not 0.  
#UD                      Operand is a register.  
                            If the LOCK prefix is used.

INVPCID—Invalidate Process-Context Identifier

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
66 OF 38 82 /r INVPCID r32, m128	RM	NE/V	INVPCID	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r32 and descriptor in m128.
66 OF 38 82 /r INVPCID r64, m128	RM	V/NE	INVPCID	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r64 and descriptor in m128.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (R)	ModRM:r/m (R)	NA	NA

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on process-context identifier (PCID). (See Section 4.10, “Caching Translation Information,” in *Intel 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A*.) Invalidation is based on the INVPCID type specified in the register operand and the INVPCID descriptor specified in the memory operand.

Outside 64-bit mode, the register operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode the register operand has 64 bits.

There are four INVPCID types currently defined:

- Individual-address invalidation: If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—for the linear address and PCID specified in the INVPCID descriptor.<sup>1</sup> In some cases, the instruction may invalidate global translations or mappings for other linear addresses (or other PCIDs) as well.
- Single-context invalidation: If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. In some cases, the instruction may invalidate global translations or mappings for other PCIDs as well.
- All-context invalidation, including global translations: If the INVPCID type is 2, the logical processor invalidates all mappings—including global translations—associated with any PCID.
- All-context invalidation: If the INVPCID type is 3, the logical processor invalidates all mappings—except global translations—associated with any PCID. In some case, the instruction may invalidate global translations as well.

The INVPCID descriptor comprises 128 bits and consists of a PCID and a linear address as shown in Figure 3-24. For INVPCID type 0, the processor uses the full 64 bits of the linear address even outside 64-bit mode; the linear address is not used for other INVPCID types.

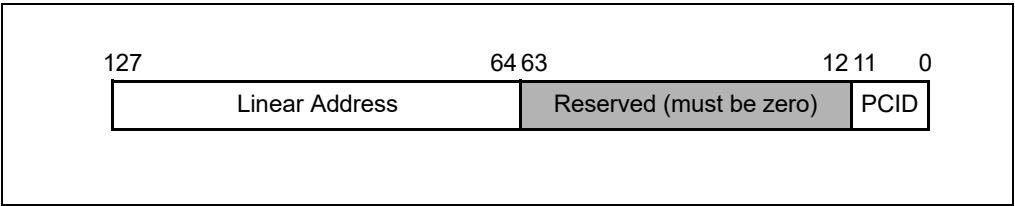


Figure 3-24. INVPCID Descriptor

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, “Details of TLB Use,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*), the instruction invalidates all of them.

If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. In this case, executions with INVPCID types 0 and 1 are allowed only if the PCID specified in the INVPCID descriptor is 000H; executions with INVPCID types 2 and 3 invalidate mappings only for PCID 000H. Note that CR4.PCIDE must be 0 outside 64-bit mode (see Chapter 4.10.1, “Process-Context Identifiers (PCIDs),” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

## Operation

```
INVPCID_TYPE ← value of register operand;      // must be in the range of 0–3
INVPCID_DESC ← value of memory operand;
CASE INVPCID_TYPE OF
  0:      // individual-address invalidation
    PCID ← INVPCID_DESC[11:0];
    L_ADDR ← INVPCID_DESC[127:64];
    Invalidate mappings for L_ADDR associated with PCID except global translations;
    BREAK;
  1:      // single PCID invalidation
    PCID ← INVPCID_DESC[11:0];
    Invalidate all mappings associated with PCID except global translations;
    BREAK;
  2:      // all PCID invalidation including global translations
    Invalidate all mappings for all PCIDs, including global translations;
    BREAK;
  3:      // all PCID invalidation retaining global translations
    Invalidate all mappings for all PCIDs except global translations;
    BREAK;
ESAC;
```

## Intel C/C++ Compiler Intrinsic Equivalent

```
INVPCID: void _invpcid(unsigned __int32 type, void * descriptor);
```

## SIMD Floating-Point Exceptions

None

## Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains an unusable segment.</p> <p>If the source operand is located in an execute-only code segment.</p> <p>If an invalid type is specified in the register operand, i.e., INVPCID_TYPE &gt; 3.</p> <p>If bits 63:12 of INVPCID_DESC are not all zero.</p> <p>If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.</p> <p>If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.</p>
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	<p>If the memory operand effective address is outside the SS segment limit.</p> <p>If the SS register contains an unusable segment.</p>
#UD	<p>If if CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.</p> <p>If the LOCK prefix is used.</p>

**Real-Address Mode Exceptions**

#GP	<p>If an invalid type is specified in the register operand, i.e., INVPCID_TYPE &gt; 3.</p> <p>If bits 63:12 of INVPCID_DESC are not all zero.</p> <p>If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.</p> <p>If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.</p>
#UD	<p>If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.</p> <p>If the LOCK prefix is used.</p>

**Virtual-8086 Mode Exceptions**

#GP(0)	The INVPCID instruction is not recognized in virtual-8086 mode.
--------	---

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.</p> <p>If an invalid type is specified in the register operand, i.e., INVPCID_TYPE &gt; 3.</p> <p>If bits 63:12 of INVPCID_DESC are not all zero.</p> <p>If CR4.PCIDE=0, INVPCID_TYPE is either 0 or 1, and INVPCID_DESC[11:0] is not zero.</p> <p>If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.</p>
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	<p>If the LOCK prefix is used.</p> <p>If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.</p>

## IRET/IRETD—Interrupt Return

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
CF	IRET	Z0	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	Z0	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	Z0	Valid	N.E.	Interrupt return (64-bit operand size).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled “Task Linking” in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

If nonmaskable interrupts (NMIs) are blocked (see Section 6.7.1, “Handling Multiple NMIs” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*), execution of the IRET instruction unblocks NMIs.

This unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.W prefix promotes operation to 64 bits (IRETQ). See the summary chart at the beginning of this section for encoding data and limits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

```

IF PE = 0
    THEN GOTO REAL-ADDRESS-MODE;
ELSIF (IA32_EFER.LMA = 0)
    THEN
        IF (EFLAGS.VM = 1)
            THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
            ELSE GOTO PROTECTED-MODE;
        FI;
    ELSE GOTO IA-32e-MODE;
FI;

REAL-ADDRESS-MODE:
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
            EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
            EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS ← Pop(); (* 16-bit pop *)
            EFLAGS[15:0] ← Pop();
        FI;
    END;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
    IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
        THEN IF OperandSize = 32
            THEN
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                EFLAGS ← Pop();
                (* VM, IOPL, VIP and VIF EFLAG bits not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            ELSE (* OperandSize = 16 *)
                EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
                CS ← Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            FI;
        ELSE

```

```

        #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    FI;
END;

PROTECTED-MODE:
    IF NT = 1
        THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
        ELSE (* OperandSize = 16 *)
            EIP ← Pop(); (* 16-bit pop; clear upper bits *)
            CS ← Pop(); (* 16-bit pop *)
            tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
        FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN GOTO RETURN-TO-VIRTUAL-8086-MODE;
    ELSE GOTO PROTECTED-MODE-RETURN;
    FI;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within CS limit
        THEN #GP(0); FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
    IF EIP not within CS limit
        THEN #GP(0); FI;
    EFLAGS ← tempEFLAGS;
    ESP ← Pop();
    SS ← Pop(); (* Pop 2 words; throw away high-order word *)
    ES ← Pop(); (* Pop 2 words; throw away high-order word *)
    DS ← Pop(); (* Pop 2 words; throw away high-order word *)
    FS ← Pop(); (* Pop 2 words; throw away high-order word *)
    GS ← Pop(); (* Pop 2 words; throw away high-order word *)
    CPL ← 3;
    (* Resume execution in Virtual-8086 mode *)
END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
    IF CS(RPL) > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
    ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF OperandSize = 32
        THEN

```



```

        ESP ← Pop();
        SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
ELSE IF OperandSize = 16
    THEN
        ESP ← Pop(); (* 16-bit pop; clear upper bits *)
        SS ← Pop(); (* 16-bit pop *)
    ELSE (* OperandSize = 64 *)
        RSP ← Pop();
        SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
FI;
IF new mode ≠ 64-Bit Mode
    THEN
        IF EIP is not within CS limit
            THEN #GP(0); FI;
        ELSE (* new mode = 64-bit mode *)
            IF RIP is non-canonical
                THEN #GP(0); FI;
FI;
EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
IF OperandSize = 32 or OperandSize = 64
    THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
IF CPL ≤ IOPL
    THEN EFLAGS(IF) ← tempEFLAGS; FI;
IF CPL = 0
    THEN
        EFLAGS(IOPL) ← tempEFLAGS;
        IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
FI;
CPL ← CS(RPL);
FOR each SegReg in (ES, FS, GS, and DS)
    DO
        tempDesc ← descriptor cache for SegReg (* hidden part of segment register *)
        IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
            THEN (* Segment register invalid *)
                SegmentSelector ← 0; (*Segment selector becomes null*)
FI;
OD;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
IF new mode ≠ 64-Bit Mode
    THEN
        IF EIP is not within CS limit
            THEN #GP(0); FI;
        ELSE (* new mode = 64-bit mode *)
            IF RIP is non-canonical
                THEN #GP(0); FI;
FI;
EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
IF OperandSize = 32 or OperandSize = 64
    THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
IF CPL ≤ IOPL
    THEN EFLAGS(IF) ← tempEFLAGS; FI;

```

```

IF CPL = 0
    THEN
        EFLAGS(IOPL) ← tempEFLAGS;
        IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
    FI;
END;

IA-32e-MODE:
IF NT = 1
    THEN #GP(0);
ELSE IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop();
        tempEFLAGS ← Pop();
    ELSE IF OperandSize = 16
        THEN
            EIP ← Pop(); (* 16-bit pop; clear upper bits *)
            CS ← Pop(); (* 16-bit pop *)
            tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
        FI;
    ELSE (* OperandSize = 64 *)
        THEN
            RIP ← Pop();
            CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
            tempRFLAGS ← Pop();
        FI;
    IF CS.RPL > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
    ELSE
        IF instruction began in 64-Bit Mode
            THEN
                IF OperandSize = 32
                    THEN
                        ESP ← Pop();
                        SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                    ELSE IF OperandSize = 16
                        THEN
                            ESP ← Pop(); (* 16-bit pop; clear upper bits *)
                            SS ← Pop(); (* 16-bit pop *)
                        ELSE (* OperandSize = 64 *)
                            RSP ← Pop();
                            SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                        FI;
                    FI;
                GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
            END;

```

### Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

**Protected Mode Exceptions**

#GP(0)	If the return code or stack segment selector is NULL. If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits. If the return code segment selector RPL is less than the CPL. If the DPL of a conforming-code segment is greater than the return code segment selector RPL. If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector. If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector. If the stack segment is not a writable data segment. If the stack segment selector RPL is not equal to the RPL of the return code segment selector. If the segment descriptor for a code segment does not indicate it is a code segment. If the segment selector for a TSS has its local/global bit set for local. If a TSS segment descriptor specifies that the TSS is not busy. If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)	If the top bytes of stack are not within stack limits. If the return stack segment is not present.
#NP (selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

**Virtual-8086 Mode Exceptions**

#GP(0)	If the return instruction pointer is not within the return code segment limit. If IOPL not equal to 3.
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

#GP(0)	If EFLAGS.NT[bit 14] = 1.
--------	---------------------------

Other exceptions same as in Protected Mode.

**64-Bit Mode Exceptions**

#GP(0)	<p>If EFLAGS.NT[bit 14] = 1.</p> <p>If the return code segment selector is NULL.</p> <p>If the stack segment selector is NULL going back to compatibility mode.</p> <p>If the stack segment selector is NULL going back to CPL3 64-bit mode.</p> <p>If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.</p> <p>If the return instruction pointer is not within the return code segment limit.</p> <p>If the return instruction pointer is non-canonical.</p>
#GP(Selector)	<p>If a segment selector index is outside its descriptor table limits.</p> <p>If a segment descriptor memory address is non-canonical.</p> <p>If the segment descriptor for a code segment does not indicate it is a code segment.</p> <p>If the proposed new code segment descriptor has both the D-bit and L-bit set.</p> <p>If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.</p> <p>If CPL is greater than the RPL of the code segment selector.</p> <p>If the DPL of a conforming-code segment is greater than the return code segment selector RPL.</p> <p>If the stack segment is not a writable data segment.</p> <p>If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.</p> <p>If the stack segment selector RPL is not equal to the RPL of the return code segment selector.</p>
#SS(0)	<p>If an attempt to pop a value off the stack violates the SS limit.</p> <p>If an attempt to pop a value off the stack causes a non-canonical address to be referenced.</p> <p>If the return stack segment is not present.</p>
#NP (selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

**Jcc—Jump if Condition Is Met**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
77 <i>cb</i>	<i>JA rel8</i>	D	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 <i>cb</i>	<i>JAE rel8</i>	D	Valid	Valid	Jump short if above or equal (CF=0).
72 <i>cb</i>	<i>JB rel8</i>	D	Valid	Valid	Jump short if below (CF=1).
76 <i>cb</i>	<i>JBE rel8</i>	D	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 <i>cb</i>	<i>JC rel8</i>	D	Valid	Valid	Jump short if carry (CF=1).
E3 <i>cb</i>	<i>JCXZ rel8</i>	D	N.E.	Valid	Jump short if CX register is 0.
E3 <i>cb</i>	<i>JECXZ rel8</i>	D	Valid	Valid	Jump short if ECX register is 0.
E3 <i>cb</i>	<i>JRCXZ rel8</i>	D	Valid	N.E.	Jump short if RCX register is 0.
74 <i>cb</i>	<i>JE rel8</i>	D	Valid	Valid	Jump short if equal (ZF=1).
7F <i>cb</i>	<i>JG rel8</i>	D	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D <i>cb</i>	<i>JGE rel8</i>	D	Valid	Valid	Jump short if greater or equal (SF=OF).
7C <i>cb</i>	<i>JL rel8</i>	D	Valid	Valid	Jump short if less (SF≠ OF).
7E <i>cb</i>	<i>JLE rel8</i>	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 <i>cb</i>	<i>JNA rel8</i>	D	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 <i>cb</i>	<i>JNAE rel8</i>	D	Valid	Valid	Jump short if not above or equal (CF=1).
73 <i>cb</i>	<i>JNB rel8</i>	D	Valid	Valid	Jump short if not below (CF=0).
77 <i>cb</i>	<i>JNBE rel8</i>	D	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).
73 <i>cb</i>	<i>JNC rel8</i>	D	Valid	Valid	Jump short if not carry (CF=0).
75 <i>cb</i>	<i>JNE rel8</i>	D	Valid	Valid	Jump short if not equal (ZF=0).
7E <i>cb</i>	<i>JNG rel8</i>	D	Valid	Valid	Jump short if not greater (ZF=1 or SF≠ OF).
7C <i>cb</i>	<i>JNGE rel8</i>	D	Valid	Valid	Jump short if not greater or equal (SF≠ OF).
7D <i>cb</i>	<i>JNL rel8</i>	D	Valid	Valid	Jump short if not less (SF=OF).
7F <i>cb</i>	<i>JNLE rel8</i>	D	Valid	Valid	Jump short if not less or equal (ZF=0 and SF=OF).
71 <i>cb</i>	<i>JNO rel8</i>	D	Valid	Valid	Jump short if not overflow (OF=0).
7B <i>cb</i>	<i>JNP rel8</i>	D	Valid	Valid	Jump short if not parity (PF=0).
79 <i>cb</i>	<i>JNS rel8</i>	D	Valid	Valid	Jump short if not sign (SF=0).
75 <i>cb</i>	<i>JNZ rel8</i>	D	Valid	Valid	Jump short if not zero (ZF=0).
70 <i>cb</i>	<i>JO rel8</i>	D	Valid	Valid	Jump short if overflow (OF=1).
7A <i>cb</i>	<i>JP rel8</i>	D	Valid	Valid	Jump short if parity (PF=1).
7A <i>cb</i>	<i>JPE rel8</i>	D	Valid	Valid	Jump short if parity even (PF=1).
7B <i>cb</i>	<i>JPO rel8</i>	D	Valid	Valid	Jump short if parity odd (PF=0).
78 <i>cb</i>	<i>JS rel8</i>	D	Valid	Valid	Jump short if sign (SF=1).
74 <i>cb</i>	<i>JZ rel8</i>	D	Valid	Valid	Jump short if zero (ZF = 1).
0F 87 <i>cw</i>	<i>JA rel16</i>	D	N.S.	Valid	Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 <i>cd</i>	<i>JA rel32</i>	D	Valid	Valid	Jump near if above (CF=0 and ZF=0).
0F 83 <i>cw</i>	<i>JAE rel16</i>	D	N.S.	Valid	Jump near if above or equal (CF=0). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 83 <i>cd</i>	JAE <i>rel32</i>	D	Valid	Valid	Jump near if above or equal (CF=0).
0F 82 <i>cw</i>	JB <i>rel16</i>	D	N.S.	Valid	Jump near if below (CF=1). Not supported in 64-bit mode.
0F 82 <i>cd</i>	JB <i>rel32</i>	D	Valid	Valid	Jump near if below (CF=1).
0F 86 <i>cw</i>	JBE <i>rel16</i>	D	N.S.	Valid	Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 <i>cd</i>	JBE <i>rel32</i>	D	Valid	Valid	Jump near if below or equal (CF=1 or ZF=1).
0F 82 <i>cw</i>	JC <i>rel16</i>	D	N.S.	Valid	Jump near if carry (CF=1). Not supported in 64-bit mode.
0F 82 <i>cd</i>	JC <i>rel32</i>	D	Valid	Valid	Jump near if carry (CF=1).
0F 84 <i>cw</i>	JE <i>rel16</i>	D	N.S.	Valid	Jump near if equal (ZF=1). Not supported in 64-bit mode.
0F 84 <i>cd</i>	JE <i>rel32</i>	D	Valid	Valid	Jump near if equal (ZF=1).
0F 84 <i>cw</i>	JZ <i>rel16</i>	D	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 <i>cd</i>	JZ <i>rel32</i>	D	Valid	Valid	Jump near if 0 (ZF=1).
0F 8F <i>cw</i>	JG <i>rel16</i>	D	N.S.	Valid	Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F <i>cd</i>	JG <i>rel32</i>	D	Valid	Valid	Jump near if greater (ZF=0 and SF=OF).
0F 8D <i>cw</i>	JGE <i>rel16</i>	D	N.S.	Valid	Jump near if greater or equal (SF=OF). Not supported in 64-bit mode.
0F 8D <i>cd</i>	JGE <i>rel32</i>	D	Valid	Valid	Jump near if greater or equal (SF=OF).
0F 8C <i>cw</i>	JL <i>rel16</i>	D	N.S.	Valid	Jump near if less (SF≠ OF). Not supported in 64-bit mode.
0F 8C <i>cd</i>	JL <i>rel32</i>	D	Valid	Valid	Jump near if less (SF≠ OF).
0F 8E <i>cw</i>	JLE <i>rel16</i>	D	N.S.	Valid	Jump near if less or equal (ZF=1 or SF≠ OF). Not supported in 64-bit mode.
0F 8E <i>cd</i>	JLE <i>rel32</i>	D	Valid	Valid	Jump near if less or equal (ZF=1 or SF≠ OF).
0F 86 <i>cw</i>	JNA <i>rel16</i>	D	N.S.	Valid	Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 <i>cd</i>	JNA <i>rel32</i>	D	Valid	Valid	Jump near if not above (CF=1 or ZF=1).
0F 82 <i>cw</i>	JNAE <i>rel16</i>	D	N.S.	Valid	Jump near if not above or equal (CF=1). Not supported in 64-bit mode.
0F 82 <i>cd</i>	JNAE <i>rel32</i>	D	Valid	Valid	Jump near if not above or equal (CF=1).
0F 83 <i>cw</i>	JNB <i>rel16</i>	D	N.S.	Valid	Jump near if not below (CF=0). Not supported in 64-bit mode.
0F 83 <i>cd</i>	JNB <i>rel32</i>	D	Valid	Valid	Jump near if not below (CF=0).
0F 87 <i>cw</i>	JNBE <i>rel16</i>	D	N.S.	Valid	Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 <i>cd</i>	JNBE <i>rel32</i>	D	Valid	Valid	Jump near if not below or equal (CF=0 and ZF=0).
0F 83 <i>cw</i>	JNC <i>rel16</i>	D	N.S.	Valid	Jump near if not carry (CF=0). Not supported in 64-bit mode.
0F 83 <i>cd</i>	JNC <i>rel32</i>	D	Valid	Valid	Jump near if not carry (CF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 85 <i>cw</i>	JNE <i>rel16</i>	D	N.S.	Valid	Jump near if not equal (ZF=0). Not supported in 64-bit mode.
0F 85 <i>cd</i>	JNE <i>rel32</i>	D	Valid	Valid	Jump near if not equal (ZF=0).
0F 8E <i>cw</i>	JNG <i>rel16</i>	D	N.S.	Valid	Jump near if not greater (ZF=1 or SF≠OF). Not supported in 64-bit mode.
0F 8E <i>cd</i>	JNG <i>rel32</i>	D	Valid	Valid	Jump near if not greater (ZF=1 or SF≠OF).
0F 8C <i>cw</i>	JNGE <i>rel16</i>	D	N.S.	Valid	Jump near if not greater or equal (SF≠OF). Not supported in 64-bit mode.
0F 8C <i>cd</i>	JNGE <i>rel32</i>	D	Valid	Valid	Jump near if not greater or equal (SF≠OF).
0F 8D <i>cw</i>	JNL <i>rel16</i>	D	N.S.	Valid	Jump near if not less (SF=OF). Not supported in 64-bit mode.
0F 8D <i>cd</i>	JNL <i>rel32</i>	D	Valid	Valid	Jump near if not less (SF=OF).
0F 8F <i>cw</i>	JNLE <i>rel16</i>	D	N.S.	Valid	Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F <i>cd</i>	JNLE <i>rel32</i>	D	Valid	Valid	Jump near if not less or equal (ZF=0 and SF=OF).
0F 81 <i>cw</i>	JNO <i>rel16</i>	D	N.S.	Valid	Jump near if not overflow (OF=0). Not supported in 64-bit mode.
0F 81 <i>cd</i>	JNO <i>rel32</i>	D	Valid	Valid	Jump near if not overflow (OF=0).
0F 8B <i>cw</i>	JNP <i>rel16</i>	D	N.S.	Valid	Jump near if not parity (PF=0). Not supported in 64-bit mode.
0F 8B <i>cd</i>	JNP <i>rel32</i>	D	Valid	Valid	Jump near if not parity (PF=0).
0F 89 <i>cw</i>	JNS <i>rel16</i>	D	N.S.	Valid	Jump near if not sign (SF=0). Not supported in 64-bit mode.
0F 89 <i>cd</i>	JNS <i>rel32</i>	D	Valid	Valid	Jump near if not sign (SF=0).
0F 85 <i>cw</i>	JNZ <i>rel16</i>	D	N.S.	Valid	Jump near if not zero (ZF=0). Not supported in 64-bit mode.
0F 85 <i>cd</i>	JNZ <i>rel32</i>	D	Valid	Valid	Jump near if not zero (ZF=0).
0F 80 <i>cw</i>	JO <i>rel16</i>	D	N.S.	Valid	Jump near if overflow (OF=1). Not supported in 64-bit mode.
0F 80 <i>cd</i>	JO <i>rel32</i>	D	Valid	Valid	Jump near if overflow (OF=1).
0F 8A <i>cw</i>	JP <i>rel16</i>	D	N.S.	Valid	Jump near if parity (PF=1). Not supported in 64-bit mode.
0F 8A <i>cd</i>	JP <i>rel32</i>	D	Valid	Valid	Jump near if parity (PF=1).
0F 8A <i>cw</i>	JPE <i>rel16</i>	D	N.S.	Valid	Jump near if parity even (PF=1). Not supported in 64-bit mode.
0F 8A <i>cd</i>	JPE <i>rel32</i>	D	Valid	Valid	Jump near if parity even (PF=1).
0F 8B <i>cw</i>	JPO <i>rel16</i>	D	N.S.	Valid	Jump near if parity odd (PF=0). Not supported in 64-bit mode.
0F 8B <i>cd</i>	JPO <i>rel32</i>	D	Valid	Valid	Jump near if parity odd (PF=0).
0F 88 <i>cw</i>	JS <i>rel16</i>	D	N.S.	Valid	Jump near if sign (SF=1). Not supported in 64-bit mode.

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 88 <i>cd</i>	<i>JS rel32</i>	D	Valid	Valid	Jump near if sign (SF=1).
OF 84 <i>cw</i>	<i>JZ rel16</i>	D	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
OF 84 <i>cd</i>	<i>JZ rel32</i>	D	Valid	Valid	Jump near if 0 (ZF=1).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA

### Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the *Jcc* instruction.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). A relative offset (*rel8*, *rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer. Instruction coding is most efficient for offsets of -128 to +127. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

The conditions for each *Jcc* mnemonic are given in the "Description" column of the table on the preceding page. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the *JA* (jump if above) instruction and the *JNBE* (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

The *Jcc* instruction does not support far jumps (jumps to other code segments). When the target for the conditional jump is in a different segment, use the opposite condition from the condition being tested for the *Jcc* instruction, and then access the target with an unconditional far jump (*JMP* instruction) to the other segment. For example, the following conditional far jump is illegal:

```
JZ FARLABEL;
```

To accomplish this far jump, use the following two instructions:

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The *JRCXZ*, *JECXZ* and *JCXZ* instructions differ from other *Jcc* instructions because they do not check status flags. Instead, they check RCX, ECX or CX for 0. The register checked is determined by the address-size attribute. These instructions are useful when used at the beginning of a loop that terminates with a conditional loop instruction (such as *LOOPNE*). They can be used to prevent an instruction sequence from entering a loop when RCX, ECX or CX is 0. This would cause the loop to execute  $2^{64}$ ,  $2^{32}$  or 64K times (not zero times).

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

In 64-bit mode, operand size is fixed at 64 bits. *JMP Short* is  $RIP = RIP + 8\text{-bit offset sign extended to 64 bits}$ . *JMP Near* is  $RIP = RIP + 32\text{-bit offset sign extended to 64 bits}$ .



Operation

```
IF condition
  THEN
    tempEIP ← EIP + SignExtend(DEST);
    IF OperandSize = 16
      THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
  IF tempEIP is not within code segment limit
    THEN #GP(0);
    ELSE EIP ← tempEIP
  FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

- #GP(0) If the offset being jumped to is beyond the limits of the CS segment.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #GP If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- #GP(0) If the memory address is in a non-canonical form.
- #UD If the LOCK prefix is used.

## JMP—Jump

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
EB <i>cb</i>	JMP <i>rel8</i>	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 <i>cw</i>	JMP <i>rel16</i>	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 <i>cd</i>	JMP <i>rel32</i>	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF /4	JMP <i>r/m16</i>	M	N.S.	Valid	Jump near, absolute indirect, address = zero-extended <i>r/m16</i> . Not supported in 64-bit mode.
FF /4	JMP <i>r/m32</i>	M	N.S.	Valid	Jump near, absolute indirect, address given in <i>r/m32</i> . Not supported in 64-bit mode.
FF /4	JMP <i>r/m64</i>	M	Valid	N.E.	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory
EA <i>cd</i>	JMP <i>ptr16:16</i>	D	Inv.	Valid	Jump far, absolute, address given in operand
EA <i>cp</i>	JMP <i>ptr16:32</i>	D	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP <i>m16:16</i>	D	Valid	Valid	Jump far, absolute indirect, address given in <i>m16:16</i>
FF /5	JMP <i>m16:32</i>	D	Valid	Valid	Jump far, absolute indirect, address given in <i>m16:32</i> .
REX.W + FF /5	JMP <i>m16:64</i>	D	Valid	N.E.	Jump far, absolute indirect, address given in <i>m16:64</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

### Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 7, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the JMP instruction).

**Near and Short Jumps.** When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current

value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (*rel8*, *rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

**Far Jumps in Real-Address or Virtual-8086 Mode.** When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

**Far Jumps in Protected Mode.** When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for detailed information on the mechanics of a task switch.

Note that when you execute at task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

**In 64-Bit Mode** — The instruction's operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```

IF near jump
  IF 64-bit Mode
    THEN
      IF near relative jump
        THEN
          tempRIP ← RIP + DEST; (* RIP is instruction following JMP instruction*)
        ELSE (* Near absolute jump *)
          tempRIP ← DEST;
      FI;
    ELSE
      IF near relative jump
        THEN
          tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
        ELSE (* Near absolute jump *)
          tempEIP ← DEST;
      FI;
    FI;
  IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
    and tempEIP outside code segment limit
    THEN #GP(0); FI
  IF 64-bit mode and tempRIP is not canonical
    THEN #GP(0);
  FI;
  IF OperandSize = 32
    THEN
      EIP ← tempEIP;
    ELSE
      IF OperandSize = 16
        THEN (* OperandSize = 16 *)
          EIP ← tempEIP AND 0000FFFFH;
        ELSE (* OperandSize = 64 *)
          RIP ← tempRIP;
      FI;
    FI;
  FI;
  IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
      tempEIP ← DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
      IF tempEIP is beyond code segment limit
        THEN #GP(0); FI;
      CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
      IF OperandSize = 32

```

```

        THEN
            EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
        ELSE (* OperandSize = 16 *)
            EIP ← tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
    FI;
FI;
IF far jump and (PE = 1 and VM = 0)
(* IA-32e mode or protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
        or segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector); FI;
        Read type and access rights of segment descriptor;
        IF (EFER.LMA = 0)
            THEN
                IF segment type is not a conforming or nonconforming code
                segment, call gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment
                call gate
                    THEN #GP(segment selector); FI;
            FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
        ELSE
            #GP(segment selector);
    FI;
CONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF DPL > CPL
        THEN #GP(segment selector); FI;
    IF segment not present
        THEN #NP(segment selector); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and
    tempEIP outside code segment limit
        THEN #GP(0); FI
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL
    EIP ← tempEIP;
END;

```

## NONCONFORMING-CODE-SEGMENT:

```

    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) OR (DPL ≠ CPL)
        THEN #GP(code segment selector); FI;
    IF segment not present
        THEN #NP(segment selector); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH; FI;
    IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
    and tempEIP outside code segment limit
        THEN #GP(0); FI;
    IF tempEIP is non-canonical THEN #GP(0); FI;
    CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL;
    EIP ← tempEIP;
END;
```

## CALL-GATE:

```

    IF call gate DPL < CPL
    or call gate DPL < call gate segment-selector RPL
        THEN #GP(call gate selector); FI;
    IF call gate not present
        THEN #NP(call gate selector); FI;
    IF call gate code-segment selector is NULL
        THEN #GP(0); FI;
    IF call gate code-segment selector index outside descriptor table limits
        THEN #GP(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
    or code-segment segment descriptor is conforming and DPL > CPL
    or code-segment segment descriptor is non-conforming and DPL ≠ CPL
        THEN #GP(code segment selector); FI;
    IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
    or code-segment segment descriptor has both L-Bit and D-bit set)
        THEN #GP(code segment selector); FI;
    IF code segment is not present
        THEN #NP(code-segment selector); FI;
    tempEIP ← DEST(Offset);
    IF GateSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH; FI;
    IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
    outside code segment limit
        THEN #GP(0); FI;
    CS ← DEST[SegmentSelector]; (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL;
    EIP ← tempEIP;
END;
```

## TASK-GATE:

```

    IF task gate DPL < CPL
    or task gate DPL < task gate segment-selector RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
```

```

        THEN #NP(gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
    or descriptor is not a TSS segment
    or TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
TASK-STATE-SEGMENT:
    IF TSS DPL < CPL
    or TSS DPL < TSS segment-selector RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

```

### Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

### Protected Mode Exceptions

#GP(0)	<p>If offset in target operand, call gate, or TSS is beyond the code segment limits.</p> <p>If the segment selector in the destination operand, call gate, task gate, or TSS is NULL.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If the segment selector index is outside descriptor table limits.</p> <p>If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.</p> <p>If the DPL for a nonconforming-code segment is not equal to the CPL (When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.</p> <p>If the DPL for a conforming-code segment is greater than the CPL.</p> <p>If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.</p> <p>If the segment descriptor for selector in a call gate does not indicate it is a code segment.</p> <p>If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.</p> <p>If the segment selector for a TSS has its local/global bit set for local.</p> <p>If a TSS segment descriptor specifies that the TSS is busy or not available.</p>
#SS(0)	<p>If a memory operand effective address is outside the SS segment limit.</p>

#NP (selector)	If the code segment being accessed is not present. If call gate, task gate, or TSS not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If the target operand is beyond the code segment limits. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

### 64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical. If target offset in destination operand is non-canonical. If target offset in destination operand is beyond the new code segment limit. If the segment selector in the destination operand is NULL. If the code segment selector in the 64-bit gate is NULL.
#GP(selector)	If the code segment or 64-bit call gate is outside descriptor table limits. If the code segment or 64-bit call gate overlaps non-canonical space. If the segment descriptor from a 64-bit call gate is in non-canonical space. If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate. If the segment descriptor pointed to by the segment selector in the destination operand is a code segment, and has both the D-bit and the L-bit set. If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL. If the DPL for a conforming-code segment is greater than the CPL. If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate. If the upper type field of a 64-bit call gate is not 0x0. If the segment selector from a 64-bit call gate is beyond the descriptor table limits. If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear. If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment. If the code segment is non-conforming and $CPL \neq DPL$ .



	If the code segment is confirming and $CPL < DPL$ .
#NP(selector)	If a code segment or 64-bit call gate is not present.
#UD	(64-bit mode only) If a far jump is direct to an absolute address in memory.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## KADDW/KADDB/KADDQ/KADD—ADD Two Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 4A /r KADDW k1, k2, k3	RVR	V/V	AVX512DQ	Add 16 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 4A /r KADDB k1, k2, k3	RVR	V/V	AVX512DQ	Add 8 bits masks in k2 and k3 and place result in k1.
VEX.L1.0F.W1 4A /r KADDQ k1, k2, k3	RVR	V/V	AVX512BW	Add 64 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 4A /r KADD k1, k2, k3	RVR	V/V	AVX512BW	Add 32 bits masks in k2 and k3 and place result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Adds the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

### Operation

#### KADDW

$$\text{DEST}[15:0] \leftarrow \text{SRC1}[15:0] + \text{SRC2}[15:0]$$

$$\text{DEST}[\text{MAX\_KL}-1:16] \leftarrow 0$$

#### KADDB

$$\text{DEST}[7:0] \leftarrow \text{SRC1}[7:0] + \text{SRC2}[7:0]$$

$$\text{DEST}[\text{MAX\_KL}-1:8] \leftarrow 0$$

#### KADDQ

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0]$$

$$\text{DEST}[\text{MAX\_KL}-1:64] \leftarrow 0$$

#### KADD

$$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0]$$

$$\text{DEST}[\text{MAX\_KL}-1:32] \leftarrow 0$$

### Intel C/C++ Compiler Intrinsic Equivalent

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type K20.

**KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 41 /r KANDW k1, k2, k3	RVR	V/V	AVX512F	Bitwise AND 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 41 /r KANDB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise AND 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 41 /r KANDQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 41 /r KANDD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND 32 bits masks k2 and k3 and place result in k1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Performs a bitwise AND between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

**Operation****KANDW**

DEST[15:0] ← SRC1[15:0] BITWISE AND SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

**KANDB**

DEST[7:0] ← SRC1[7:0] BITWISE AND SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

**KANDQ**

DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

**KANDD**

DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KANDW \_\_mmask16 \_mm512\_kand(\_\_mmask16 a, \_\_mmask16 b);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 42 /r KANDNW k1, k2, k3	RVR	V/V	AVX512F	Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 42 /r KANDNB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise AND NOT 8 bits masks k1 and k2 and place result in k1.
VEX.L1.0F.W1 42 /r KANDNQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND NOT 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 42 /r KANDND k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND NOT 32 bits masks k2 and k3 and place result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Performs a bitwise AND NOT between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

### Operation

#### KANDNW

DEST[15:0] ← (BITWISE NOT SRC1[15:0]) BITWISE AND SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

#### KANDNB

DEST[7:0] ← (BITWISE NOT SRC1[7:0]) BITWISE AND SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

#### KANDNQ

DEST[63:0] ← (BITWISE NOT SRC1[63:0]) BITWISE AND SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

#### KANDND

DEST[31:0] ← (BITWISE NOT SRC1[31:0]) BITWISE AND SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

KANDNW \_\_mmask16\_mm512\_kandn(\_\_mmask16 a, \_\_mmask16 b);

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type K20.

**KMOVW/KMOVB/KMOVQ/KMOVD—Move from and to Mask Registers**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 90 /r KMOVW k1, k2/m16	RM	V/V	AVX512F	Move 16 bits mask from k2/m16 and store the result in k1.
VEX.L0.66.0F.W0 90 /r KMOVB k1, k2/m8	RM	V/V	AVX512DQ	Move 8 bits mask from k2/m8 and store the result in k1.
VEX.L0.0F.W1 90 /r KMOVQ k1, k2/m64	RM	V/V	AVX512BW	Move 64 bits mask from k2/m64 and store the result in k1.
VEX.L0.66.0F.W1 90 /r KMOVD k1, k2/m32	RM	V/V	AVX512BW	Move 32 bits mask from k2/m32 and store the result in k1.
VEX.L0.0F.W0 91 /r KMOVW m16, k1	MR	V/V	AVX512F	Move 16 bits mask from k1 and store the result in m16.
VEX.L0.66.0F.W0 91 /r KMOVB m8, k1	MR	V/V	AVX512DQ	Move 8 bits mask from k1 and store the result in m8.
VEX.L0.0F.W1 91 /r KMOVQ m64, k1	MR	V/V	AVX512BW	Move 64 bits mask from k1 and store the result in m64.
VEX.L0.66.0F.W1 91 /r KMOVD m32, k1	MR	V/V	AVX512BW	Move 32 bits mask from k1 and store the result in m32.
VEX.L0.0F.W0 92 /r KMOVW k1, r32	RR	V/V	AVX512F	Move 16 bits mask from r32 to k1.
VEX.L0.66.0F.W0 92 /r KMOVB k1, r32	RR	V/V	AVX512DQ	Move 8 bits mask from r32 to k1.
VEX.L0.F2.0F.W1 92 /r KMOVQ k1, r64	RR	V/I	AVX512BW	Move 64 bits mask from r64 to k1.
VEX.L0.F2.0F.W0 92 /r KMOVD k1, r32	RR	V/V	AVX512BW	Move 32 bits mask from r32 to k1.
VEX.L0.0F.W0 93 /r KMOVW r32, k1	RR	V/V	AVX512F	Move 16 bits mask from k1 to r32.
VEX.L0.66.0F.W0 93 /r KMOVB r32, k1	RR	V/V	AVX512DQ	Move 8 bits mask from k1 to r32.
VEX.L0.F2.0F.W1 93 /r KMOVQ r64, k1	RR	V/I	AVX512BW	Move 64 bits mask from k1 to r64.
VEX.L0.F2.0F.W0 93 /r KMOVD r32, k1	RR	V/V	AVX512BW	Move 32 bits mask from k1 to r32.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2
RM	ModRM:reg (w)	ModRM:r/m (r)
MR	ModRM:r/m (w, ModRM:[7:6] must not be 11b)	ModRM:reg (r)
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Copies values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be mask registers, memory location or general purpose. The instruction cannot be used to transfer data between general purpose registers and or memory locations.

When moving to a mask register, the result is zero extended to MAX\_KL size (i.e., 64 bits currently). When moving to a general-purpose register (GPR), the result is zero-extended to the size of the destination. In 32-bit mode, the default GPR destination's size is 32 bits. In 64-bit mode, the default GPR destination's size is 64 bits. Note that VEX.W can only be used to modify the size of the GPR operand in 64b mode.

## Operation

### KMOVW

IF \*destination is a memory location\*

$$\text{DEST}[15:0] \leftarrow \text{SRC}[15:0]$$

IF \*destination is a mask register or a GPR \*

$$\text{DEST} \leftarrow \text{ZeroExtension}(\text{SRC}[15:0])$$

### KMOVB

IF \*destination is a memory location\*

$$\text{DEST}[7:0] \leftarrow \text{SRC}[7:0]$$

IF \*destination is a mask register or a GPR \*

$$\text{DEST} \leftarrow \text{ZeroExtension}(\text{SRC}[7:0])$$

### KMOVQ

IF \*destination is a memory location or a GPR\*

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$

IF \*destination is a mask register\*

$$\text{DEST} \leftarrow \text{ZeroExtension}(\text{SRC}[63:0])$$

### KMOVD

IF \*destination is a memory location\*

$$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$$

IF \*destination is a mask register or a GPR \*

$$\text{DEST} \leftarrow \text{ZeroExtension}(\text{SRC}[31:0])$$

## Intel C/C++ Compiler Intrinsic Equivalent

KMOVW `__mmask16 _mm512_kmov(__mmask16 a);`

## Flags Affected

None

## SIMD Floating-Point Exceptions

None

## Other Exceptions

Instructions with RR operand encoding See Exceptions Type K20.

Instructions with RM or MR operand encoding See Exceptions Type K21.

**KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 44 /r KNOTW k1, k2	RR	V/V	AVX512F	Bitwise NOT of 16 bits mask k2.
VEX.L0.66.0F.W0 44 /r KNOTB k1, k2	RR	V/V	AVX512DQ	Bitwise NOT of 8 bits mask k2.
VEX.L0.0F.W1 44 /r KNOTQ k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 64 bits mask k2.
VEX.L0.66.0F.W1 44 /r KNOTD k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 32 bits mask k2.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Performs a bitwise NOT of vector mask k2 and writes the result into vector mask k1.

**Operation****KNOTW**

DEST[15:0] ← BITWISE NOT SRC[15:0]

DEST[MAX\_KL-1:16] ← 0

**KNOTB**

DEST[7:0] ← BITWISE NOT SRC[7:0]

DEST[MAX\_KL-1:8] ← 0

**KNOTQ**

DEST[63:0] ← BITWISE NOT SRC[63:0]

DEST[MAX\_KL-1:64] ← 0

**KNOTD**

DEST[31:0] ← BITWISE NOT SRC[31:0]

DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KNOTW \_\_mmask16 \_mm512\_knot(\_\_mmask16 a);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 45 /r KORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise OR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 45 /r KORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise OR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 45 /r KORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise OR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 45 /r KORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise OR 32 bits masks k2 and k3 and place result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Performs a bitwise OR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

### Operation

#### KORW

DEST[15:0] ← SRC1[15:0] BITWISE OR SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

#### KORB

DEST[7:0] ← SRC1[7:0] BITWISE OR SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

#### KORQ

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

#### KORD

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

KORW \_\_mmask16 \_mm512\_kor(\_\_mmask16 a, \_\_mmask16 b);

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type K20.



**KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks And Set Flags**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 98 /r KORTESTW k1, k2	RR	V/V	AVX512F	Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W0 98 /r KORTESTB k1, k2	RR	V/V	AVX512DQ	Bitwise OR 8 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.0F.W1 98 /r KORTESTQ k1, k2	RR	V/V	AVX512BW	Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W1 98 /r KORTESTD k1, k2	RR	V/V	AVX512BW	Bitwise OR 32 bits masks k1 and k2 and update ZF and CF accordingly.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Performs a bitwise OR between the vector mask register k2, and the vector mask register k1, and sets CF and ZF based on the operation result.

ZF flag is set if both sources are 0x0. CF is set if, after the OR operation is done, the operation result is all 1's.

**Operation****KORTESTW**

TMP[15:0] ← DEST[15:0] BITWISE OR SRC[15:0]

IF(TMP[15:0]=0)

THEN ZF ← 1

ELSE ZF ← 0

FI;

IF(TMP[15:0]=FFFFh)

THEN CF ← 1

ELSE CF ← 0

FI;

**KORTESTB**

TMP[7:0] ← DEST[7:0] BITWISE OR SRC[7:0]

IF(TMP[7:0]=0)

THEN ZF ← 1

ELSE ZF ← 0

FI;

IF(TMP[7:0]==FFh)

THEN CF ← 1

ELSE CF ← 0

FI;

**KORTESTQ**

$\text{TMP}[63:0] \leftarrow \text{DEST}[63:0] \text{ BITWISE OR } \text{SRC}[63:0]$

IF( $\text{TMP}[63:0]=0$ )

    THEN  $\text{ZF} \leftarrow 1$

    ELSE  $\text{ZF} \leftarrow 0$

FI;

IF( $\text{TMP}[63:0] \neq \text{FFFFFFFF\_FFFFFFFFh}$ )

    THEN  $\text{CF} \leftarrow 1$

    ELSE  $\text{CF} \leftarrow 0$

FI;

**KORTESTD**

$\text{TMP}[31:0] \leftarrow \text{DEST}[31:0] \text{ BITWISE OR } \text{SRC}[31:0]$

IF( $\text{TMP}[31:0]=0$ )

    THEN  $\text{ZF} \leftarrow 1$

    ELSE  $\text{ZF} \leftarrow 0$

FI;

IF( $\text{TMP}[31:0] \neq \text{FFFFFFFFh}$ )

    THEN  $\text{CF} \leftarrow 1$

    ELSE  $\text{CF} \leftarrow 0$

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

`KORTESTW __mmask16 _mm512_kortest[cz](__mmask16 a, __mmask16 b);`

**Flags Affected**

The ZF flag is set if the result of OR-ing both sources is all 0s.

The CF flag is set if the result of OR-ing both sources is all 1s.

The OF, SF, AF, and PF flags are set to 0.

**Other Exceptions**

See Exceptions Type K20.

**KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 32 /r KSHIFTLW k1, k2, imm8	RRI	V/V	AVX512F	Shift left 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 32 /r KSHIFTLB k1, k2, imm8	RRI	V/V	AVX512DQ	Shift left 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 33 /r KSHIFTLQ k1, k2, imm8	RRI	V/V	AVX512BW	Shift left 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 33 /r KSHIFTLD k1, k2, imm8	RRI	V/V	AVX512BW	Shift left 32 bits in k2 by immediate and write result in k1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	Imm8

**Description**

Shifts 8/16/32/64 bits in the second operand (source operand) left by the count specified in immediate byte and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

**Operation****KSHIFTLW**

```

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <= 15
    THEN DEST[15:0] ← SRC1[15:0] << COUNT;
FI;
```

**KSHIFTLB**

```

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <= 7
    THEN DEST[7:0] ← SRC1[7:0] << COUNT;
FI;
```

**KSHIFTLQ**

```

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <= 63
    THEN DEST[63:0] ← SRC1[63:0] << COUNT;
FI;
```

**KSHIFTLD**

COUNT  $\leftarrow$  imm8[7:0]

DEST[MAX\_KL-1:0]  $\leftarrow$  0

IF COUNT  $\leq$  31

    THEN     DEST[31:0]  $\leftarrow$  SRC1[31:0]  $\ll$  COUNT;  
FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

Compiler auto generates KSHIFTLW when needed.

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

**KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 30 /r KSHIFTRW k1, k2, imm8	RRI	V/V	AVX512F	Shift right 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 30 /r KSHIFTRB k1, k2, imm8	RRI	V/V	AVX512DQ	Shift right 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 31 /r KSHIFTRQ k1, k2, imm8	RRI	V/V	AVX512BW	Shift right 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 31 /r KSHIFTRD k1, k2, imm8	RRI	V/V	AVX512BW	Shift right 32 bits in k2 by immediate and write result in k1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	Imm8

**Description**

Shifts 8/16/32/64 bits in the second operand (source operand) right by the count specified in immediate and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

**Operation****KSHIFTRW**

```

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <= 15
    THEN DEST[15:0] ← SRC1[15:0] >> COUNT;
FI;
```

**KSHIFTRB**

```

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <= 7
    THEN DEST[7:0] ← SRC1[7:0] >> COUNT;
FI;
```

**KSHIFTRQ**

```

COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <= 63
    THEN DEST[63:0] ← SRC1[63:0] >> COUNT;
FI;
```

**KSHIFTRD**

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT ≤ 31
    THEN DEST[31:0] ← SRC1[31:0] >> COUNT;
FI;
```

**Intel C/C++ Compiler Intrinsic Equivalent**

Compiler auto generates KSHIFTRW when needed.

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

**KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags**

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 99 /r KTESTW k1, k2	RR	V/V	AVX512DQ	Set ZF and CF depending on sign bit AND and ANDN of 16 bits mask register sources.
VEX.L0.66.0F.W0 99 /r KTESTB k1, k2	RR	V/V	AVX512DQ	Set ZF and CF depending on sign bit AND and ANDN of 8 bits mask register sources.
VEX.L0.0F.W1 99 /r KTESTQ k1, k2	RR	V/V	AVX512BW	Set ZF and CF depending on sign bit AND and ANDN of 64 bits mask register sources.
VEX.L0.66.0F.W1 99 /r KTESTD k1, k2	RR	V/V	AVX512BW	Set ZF and CF depending on sign bit AND and ANDN of 32 bits mask register sources.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand2
RR	ModRM:reg (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Performs a bitwise comparison of the bits of the first source operand and corresponding bits in the second source operand. If the AND operation produces all zeros, the ZF is set else the ZF is clear. If the bitwise AND operation of the inverted first source operand with the second source operand produces all zeros the CF is set else the CF is clear. Only the EFLAGS register is updated.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation****KTESTW**

TEMP[15:0] ← SRC2[15:0] AND SRC1[15:0]

IF (TEMP[15:0] == 0)

THEN ZF ← 1;

ELSE ZF ← 0;

FI;

TEMP[15:0] ← SRC2[15:0] AND NOT SRC1[15:0]

IF (TEMP[15:0] == 0)

THEN CF ← 1;

ELSE CF ← 0;

FI;

AF ← OF ← PF ← SF ← 0;

**KTESTB**

TEMP[7:0] ← SRC2[7:0] AND SRC1[7:0]

IF (TEMP[7:0] == 0)

THEN ZF ← 1;

ELSE ZF ← 0;

FI;

TEMP[7:0] ← SRC2[7:0] AND NOT SRC1[7:0]

IF (TEMP[7:0] == 0)

THEN CF ← 1;

ELSE CF ← 0;

FI;

AF ← OF ← PF ← SF ← 0;

**KTESTQ**

TEMP[63:0]  $\leftarrow$  SRC2[63:0] AND SRC1[63:0]

IF (TEMP[63:0] == 0)

    THEN ZF  $\leftarrow$  1;

    ELSE ZF  $\leftarrow$  0;

FI;

TEMP[63:0]  $\leftarrow$  SRC2[63:0] AND NOT SRC1[63:0]

IF (TEMP[63:0] == 0)

    THEN CF  $\leftarrow$  1;

    ELSE CF  $\leftarrow$  0;

FI;

AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;

**KTESTD**

TEMP[31:0]  $\leftarrow$  SRC2[31:0] AND SRC1[31:0]

IF (TEMP[31:0] == 0)

    THEN ZF  $\leftarrow$  1;

    ELSE ZF  $\leftarrow$  0;

FI;

TEMP[31:0]  $\leftarrow$  SRC2[31:0] AND NOT SRC1[31:0]

IF (TEMP[31:0] == 0)

    THEN CF  $\leftarrow$  1;

    ELSE CF  $\leftarrow$  0;

FI;

AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;

**Intel C/C++ Compiler Intrinsic Equivalent****SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.



## KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.66.0F.W0 4B /r KUNPCKBW k1, k2, k3	RVR	V/V	AVX512F	Unpack and interleave 8 bits masks in k2 and k3 and write word result in k1.
VEX.NDS.L1.0F.W0 4B /r KUNPCKWD k1, k2, k3	RVR	V/V	AVX512BW	Unpack and interleave 16 bits in k2 and k3 and write double-word result in k1.
VEX.NDS.L1.0F.W1 4B /r KUNPCKDQ k1, k2, k3	RVR	V/V	AVX512BW	Unpack and interleave 32 bits masks in k2 and k3 and write quadword result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Unpacks the lower 8/16/32 bits of the second and third operands (source operands) into the low part of the first operand (destination operand), starting from the low bytes. The result is zero-extended in the destination.

### Operation

#### KUNPCKBW

```
DEST[7:0] ← SRC2[7:0]
DEST[15:8] ← SRC1[7:0]
DEST[MAX_KL-1:16] ← 0
```

#### KUNPCKWD

```
DEST[15:0] ← SRC2[15:0]
DEST[31:16] ← SRC1[15:0]
DEST[MAX_KL-1:32] ← 0
```

#### KUNPCKDQ

```
DEST[31:0] ← SRC2[31:0]
DEST[63:32] ← SRC1[31:0]
DEST[MAX_KL-1:64] ← 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
KUNPCKBW __mmask16 _mm512_kunpackb(__mmask16 a, __mmask16 b);
KUNPCKDQ __mmask64 _mm512_kunpackd(__mmask64 a, __mmask64 b);
KUNPCKWD __mmask32 _mm512_kunpackw(__mmask32 a, __mmask32 b);
```

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type K20.

## KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 46 /r KXNORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise XNOR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 46 /r KXNORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise XNOR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 46 /r KXNORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XNOR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 46 /r KXNORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XNOR 32 bits masks k2 and k3 and place result in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

### Description

Performs a bitwise XNOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

### Operation

#### KXNORW

DEST[15:0] ← NOT (SRC1[15:0] BITWISE XOR SRC2[15:0])  
DEST[MAX\_KL-1:16] ← 0

#### KXNORB

DEST[7:0] ← NOT (SRC1[7:0] BITWISE XOR SRC2[7:0])  
DEST[MAX\_KL-1:8] ← 0

#### KXNORQ

DEST[63:0] ← NOT (SRC1[63:0] BITWISE XOR SRC2[63:0])  
DEST[MAX\_KL-1:64] ← 0

#### KXNORD

DEST[31:0] ← NOT (SRC1[31:0] BITWISE XOR SRC2[31:0])  
DEST[MAX\_KL-1:32] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

KXNORW \_\_mmask16 \_mm512\_kxnor(\_\_mmask16 a, \_\_mmask16 b);

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type K20.

**KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 47 /r KXORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise XOR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 47 /r KXORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise XOR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 47 /r KXORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XOR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 47 /r KXORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XOR 32 bits masks k2 and k3 and place result in k1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1 vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

**Description**

Performs a bitwise XOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

**Operation****KXORW**

DEST[15:0] ← SRC1[15:0] BITWISE XOR SRC2[15:0]  
DEST[MAX\_KL-1:16] ← 0

**KXORB**

DEST[7:0] ← SRC1[7:0] BITWISE XOR SRC2[7:0]  
DEST[MAX\_KL-1:8] ← 0

**KXORQ**

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]  
DEST[MAX\_KL-1:64] ← 0

**KXORD**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]  
DEST[MAX\_KL-1:32] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

KXORW \_\_mmask16 \_\_mm512\_xor(\_\_mmask16 a, \_\_mmask16 b);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type K20.

## LAHF—Load Status Flags into AH Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9F	LAHF	Z0	Invalid*	Valid	Load: AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF).

### NOTES:

\*Valid in specific steppings. See Description section.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

### Operation

```

IF 64-Bit Mode
  THEN
    IF CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1;
      THEN AH ← RFLAGS(SF:ZF:0:AF:0:PF:1:CF);
      ELSE #UD;
    FI;
  ELSE
    AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
  FI;

```

### Flags Affected

None. The state of the flags in the EFLAGS register is not affected.

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#UD If CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 0.  
If the LOCK prefix is used.

## LAR—Load Access Rights Byte

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 02 /r	LAR <i>r16</i> , <i>r16/m16</i>	RM	Valid	Valid	<i>r16</i> ← access rights referenced by <i>r16/m16</i>
OF 02 /r	LAR <i>reg</i> , <i>r32/m16</i> <sup>1</sup>	RM	Valid	Valid	<i>reg</i> ← access rights referenced by <i>r32/m16</i>

### NOTES:

1. For all loads (regardless of source or destination sizing) only bits 16-0 are used. Other bits are ignored.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the flag register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

The access rights for a segment descriptor include fields located in the second doubleword (bytes 4–7) of the segment descriptor. The following fields are loaded by the LAR instruction:

- Bits 7:0 are returned as 0
- Bits 11:8 return the segment type.
- Bit 12 returns the S flag.
- Bits 14:13 return the DPL.
- Bit 15 returns the P flag.
- The following fields are returned only if the operand size is greater than 16 bits:
  - Bits 19:16 are undefined.
  - Bit 20 returns the software-available bit in the descriptor.
  - Bit 21 returns the L flag.
  - Bit 22 returns the D/B flag.
  - Bit 23 returns the G flag.
  - Bits 31:24 are returned as 0.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-52.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode and IA-32e mode.

**Table 3-52. Segment and Gate Types**

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	Yes	Reserved	No
5	16-bit/32-bit task gate	Yes	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	Available 64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	Yes	64-bit call gate	Yes
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

### Operation

IF Offset(SRC) > descriptor table limit

THEN

ZF ← 0;

ELSE

SegmentDescriptor ← descriptor referenced by SRC;

IF SegmentDescriptor(Type) ≠ conforming code segment

and (CPL > DPL) or (RPL > DPL)

or SegmentDescriptor(Type) is not valid for instruction

THEN

ZF ← 0;

ELSE

DEST ← access rights from SegmentDescriptor as given in Description section;

ZF ← 1;

FI;

FI;

### Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD	The LAR instruction is not recognized in real-address mode.
-----	---

**Virtual-8086 Mode Exceptions**

#UD	The LAR instruction cannot be executed in virtual-8086 mode.
-----	--

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory operand effective address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## LDDQU—Load Unaligned Integer 128 Bits

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F F0 /r LDDQU <i>xmm1</i> , <i>mem</i>	RM	V/V	SSE3	Load unaligned data from <i>mem</i> and return double quadword in <i>xmm1</i> .
VEX.128.F2.0F.WIG F0 /r VLDDQU <i>xmm1</i> , <i>m128</i>	RM	V/V	AVX	Load unaligned packed integer values from <i>mem</i> to <i>xmm1</i> .
VEX.256.F2.0F.WIG F0 /r VLDDQU <i>ymm1</i> , <i>m256</i>	RM	V/V	AVX	Load unaligned packed integer values from <i>mem</i> to <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

The instruction is *functionally similar* to (V)MOVDQU *ymm/xmm*, *m256/m128* for loading from memory. That is: 32/16 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 32/16-byte boundary. Up to 64/32 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to (V)MOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by (V)LDDQU be modified and stored to the same location, use (V)MOVDQU or (V)MOVDQA instead of (V)LDDQU. To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the (V)MOVDQA instruction.

### Implementation Notes

- If the source is aligned to a 32/16-byte boundary, based on the implementation, the 32/16 bytes may be loaded more than once. For that reason, the usage of (V)LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using (V)MOVDQU.
- This instruction is a replacement for (V)MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use (V)MOVDQA store-load pairs when data is 256/128-bit aligned or (V)MOVDQU store-load pairs when data is 256/128-bit unaligned.
- If the memory address is not aligned on 32/16-byte boundary, some implementations may load up to 64/32 bytes and return 32/16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32/16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.
- If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the memory address is not aligned on an 8-byte boundary.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### LDDQU (128-bit Legacy SSE version)

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)



**VLDDQU (VEX.128 encoded version)**

DEST[127:0]  $\leftarrow$  SRC[127:0]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VLDDQU (VEX.256 encoded version)**

DEST[255:0]  $\leftarrow$  SRC[255:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

LDDQU: `__m128i _mm_lddqu_si128 (__m128i * p);`

VLDDQU: `__m256i _mm256_lddqu_si256 (__m256i * p);`

**Numeric Exceptions**

None

**Other Exceptions**

See Exceptions Type 4;

Note treatment of #AC varies.

## LDMXCSR—Load MXCSR Register

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP OF AE /2 LDMXCSR <i>m32</i>	M	V/V	SSE	Load MXCSR register from <i>m32</i> .
VEX.LZ.OF.WIG AE /2 VLDMXCSR <i>m32</i>	M	V/V	AVX	Load MXCSR register from <i>m32</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See “MXCSR Control and Status Register” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for a description of the MXCSR register and its contents.

The LDMXCSR instruction is typically used in conjunction with the (V)STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 1F80H.

If a (V)LDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

- the instruction must operate on an XMM or YMM register operand,
- the instruction causes that particular SIMD floating-point exception to be reported.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

If VLDMXCSR is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

$\text{MXCSR} \leftarrow m32;$

### C/C++ Compiler Intrinsic Equivalent

`_mm_setcsr(unsigned int i)`

### Numeric Exceptions

None

### Other Exceptions

See Exceptions Type 5; additionally

#GP For an attempt to set reserved bits in MXCSR.

#UD If VEX.vvvv ≠ 1111B.

## LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C5 /r	LDS r16,m16:16	RM	Invalid	Valid	Load DS:r16 with far pointer from memory.
C5 /r	LDS r32,m16:32	RM	Invalid	Valid	Load DS:r32 with far pointer from memory.
OF B2 /r	LSS r16,m16:16	RM	Valid	Valid	Load SS:r16 with far pointer from memory.
OF B2 /r	LSS r32,m16:32	RM	Valid	Valid	Load SS:r32 with far pointer from memory.
REX + OF B2 /r	LSS r64,m16:64	RM	Valid	N.E.	Load SS:r64 with far pointer from memory.
C4 /r	LES r16,m16:16	RM	Invalid	Valid	Load ES:r16 with far pointer from memory.
C4 /r	LES r32,m16:32	RM	Invalid	Valid	Load ES:r32 with far pointer from memory.
OF B4 /r	LFS r16,m16:16	RM	Valid	Valid	Load FS:r16 with far pointer from memory.
OF B4 /r	LFS r32,m16:32	RM	Valid	Valid	Load FS:r32 with far pointer from memory.
REX + OF B4 /r	LFS r64,m16:64	RM	Valid	N.E.	Load FS:r64 with far pointer from memory.
OF B5 /r	LGS r16,m16:16	RM	Valid	Valid	Load GS:r16 with far pointer from memory.
OF B5 /r	LGS r32,m16:32	RM	Valid	Valid	Load GS:r32 with far pointer from memory.
REX + OF B5 /r	LGS r64,m16:64	RM	Valid	N.E.	Load GS:r64 with far pointer from memory.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

## Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a NULL selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a NULL selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.W promotes operation to specify a source operand referencing an 80-bit pointer (16-bit selector, 64-bit offset) in memory. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). See the summary chart at the beginning of this section for encoding data and limits.

## Operation

64-BIT\_MODE

IF SS is loaded

THEN

IF SegmentSelector = NULL and ( (RPL = 3) or  
(RPL ≠ 3 and RPL ≠ CPL) )

THEN #GP(0);

ELSE IF descriptor is in non-canonical space

```

        THEN #GP(0); FI;
    ELSE IF Segment selector index is not within descriptor table limits
        or segment selector RPL  $\neq$  CPL
        or access rights indicate nonwritable data segment
        or DPL  $\neq$  CPL
        THEN #GP(selector); FI;
    ELSE IF Segment marked not present
        THEN #SS(selector); FI;
    FI;
    SS  $\leftarrow$  SegmentSelector(SRC);
    SS  $\leftarrow$  SegmentDescriptor([SRC]);
ELSE IF attempt to load DS, or ES
    THEN #UD;
ELSE IF FS, or GS is loaded with non-NULL segment selector
    THEN IF Segment selector index is not within descriptor table limits
        or access rights indicate segment neither data nor readable code segment
        or segment is data or nonconforming-code segment
        and ( RPL > DPL or CPL > DPL)
        THEN #GP(selector); FI;
    ELSE IF Segment marked not present
        THEN #NP(selector); FI;
    FI;
    SegmentRegister  $\leftarrow$  SegmentSelector(SRC) ;
    SegmentRegister  $\leftarrow$  SegmentDescriptor([SRC]);
FI;
ELSE IF FS, or GS is loaded with a NULL selector:
    THEN
        SegmentRegister  $\leftarrow$  NULLSelector;
        SegmentRegister(DescriptorValidBit)  $\leftarrow$  0; FI; (* Hidden flag;
            not accessible by software *)
FI;
DEST  $\leftarrow$  Offset(SRC);

PREOTECTED MODE OR COMPATIBILITY MODE;
IF SS is loaded
    THEN
        IF SegementSelector = NULL
            THEN #GP(0);
        ELSE IF Segment selector index is not within descriptor table limits
            or segment selector RPL  $\neq$  CPL
            or access rights indicate nonwritable data segment
            or DPL  $\neq$  CPL
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
            THEN #SS(selector); FI;
        FI;
        SS  $\leftarrow$  SegmentSelector(SRC);
        SS  $\leftarrow$  SegmentDescriptor([SRC]);
    ELSE IF DS, ES, FS, or GS is loaded with non-NULL segment selector
        THEN IF Segment selector index is not within descriptor table limits
            or access rights indicate segment neither data nor readable code segment
            or segment is data or nonconforming-code segment
            and (RPL > DPL or CPL > DPL)
            THEN #GP(selector); FI;

```

```

        ELSE IF Segment marked not present
            THEN #NP(selector); FI;
        FI;
        SegmentRegister ← SegmentSelector(SRC) AND RPL;
        SegmentRegister ← SegmentDescriptor([SRC]);
    FI;
    ELSE IF DS, ES, FS, or GS is loaded with a NULL selector:
        THEN
            SegmentRegister ← NULLSelector;
            SegmentRegister(DescriptorValidBit) ← 0; FI; (* Hidden flag;
                not accessible by software *)
    FI;
    DEST ← Offset(SRC);

Real-Address or Virtual-8086 Mode
    SegmentRegister ← SegmentSelector(SRC); FI;
    DEST ← Offset(SRC);

```

### Flags Affected

None

### Protected Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.
#GP(0)	If a NULL selector is loaded into the SS register. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL. If the DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If source operand is not a memory location. If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If a NULL selector is attempted to be loaded into the SS register in compatibility mode. If a NULL selector is attempted to be loaded into the SS register in CPL3 and 64-bit mode. If a NULL selector is attempted to be loaded into the SS register in non-CPL3 and 64-bit mode where its RPL is not equal to CPL.
#GP(Selector)	If the FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the memory address of the descriptor is non-canonical, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL. If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the memory address of the descriptor is non-canonical, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.
#SS(0)	If a memory operand effective address is non-canonical
#SS(Selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If source operand is not a memory location. If the LOCK prefix is used.

## LEA—Load Effective Address

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
8D /r	LEA r16,m	RM	Valid	Valid	Store effective address for <i>m</i> in register <i>r16</i> .
8D /r	LEA r32,m	RM	Valid	Valid	Store effective address for <i>m</i> in register <i>r32</i> .
REX.W + 8D /r	LEA r64,m	RM	Valid	N.E.	Store effective address for <i>m</i> in register <i>r64</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

**Table 3-53. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes**

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

In 64-bit mode, the instruction's destination operand is governed by operand size attribute, the default operand size is 32 bits. Address calculation is governed by address size attribute, the default address size is 64-bits. In 64-bit mode, address size of 16 bits is not encodable. See Table 3-54.

**Table 3-54. 64-bit Mode LEA Operation with Address and Operand Size Attributes**

Operand Size	Address Size	Action Performed
16	32	32-bit effective address is calculated (using 67H prefix). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
16	64	64-bit effective address is calculated (default address size). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
32	32	32-bit effective address is calculated (using 67H prefix) and stored in the requested 32-bit register destination.
32	64	64-bit effective address is calculated (default address size) and the lower 32 bits of the address are stored in the requested 32-bit register destination.
64	32	32-bit effective address is calculated (using 67H prefix), zero-extended to 64-bits, and stored in the requested 64-bit register destination (using REX.W).
64	64	64-bit effective address is calculated (default address size) and all 64-bits of the address are stored in the requested 64-bit register destination (using REX.W).

## Operation

```

IF OperandSize = 16 and AddressSize = 16
    THEN
        DEST ← EffectiveAddress(SRC); (* 16-bit address *)
    ELSE IF OperandSize = 16 and AddressSize = 32
        THEN
            temp ← EffectiveAddress(SRC); (* 32-bit address *)
            DEST ← temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 16
        THEN
            temp ← EffectiveAddress(SRC); (* 16-bit address *)
            DEST ← ZeroExtend(temp); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 32
        THEN
            DEST ← EffectiveAddress(SRC); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 16 and AddressSize = 64
        THEN
            temp ← EffectiveAddress(SRC); (* 64-bit address *)
            DEST ← temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 64
        THEN
            temp ← EffectiveAddress(SRC); (* 64-bit address *)
            DEST ← temp[0:31]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 64 and AddressSize = 64
        THEN
            DEST ← EffectiveAddress(SRC); (* 64-bit address *)
        FI;
    FI;

```

## Flags Affected

None

## Protected Mode Exceptions

#UD                      If source operand is not a memory location.  
                             If the LOCK prefix is used.

## Real-Address Mode Exceptions

Same exceptions as in protected mode.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

Same exceptions as in protected mode.



## LEAVE—High Level Procedure Exit

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C9	LEAVE	Z0	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	Z0	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	Z0	Valid	N.E.	Set RSP to RBP, then pop RBP.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See “Procedure Calls for Block-Structured Languages” in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on the use of the ENTER and LEAVE instructions.

In 64-bit mode, the instruction's default operation size is 64 bits; 32-bit operation cannot be encoded. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
IF StackAddressSize = 32
    THEN
        ESP ← EBP;
    ELSE IF StackAddressSize = 64
        THEN RSP ← RBP; FI;
    ELSE IF StackAddressSize = 16
        THEN SP ← BP; FI;
FI;
```

```
IF OperandSize = 32
    THEN EBP ← Pop();
    ELSE IF OperandSize = 64
        THEN RBP ← Pop(); FI;
    ELSE IF OperandSize = 16
        THEN BP ← Pop(); FI;
FI;
```

### Flags Affected

None

**Protected Mode Exceptions**

#SS(0)	If the EBP register points to a location that is not within the limits of the current stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If the stack address is in a non-canonical form.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## LFENCE—Load Fence

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP 0F AE E8	LFENCE	Z0	Valid	Valid	Serializes load operations.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. In particular, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. (An LFENCE that follows an instruction that stores to memory might complete before the data being stored have become globally visible.) Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute until the LFENCE completes.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue and speculative reads. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The LFENCE instruction provides a performance-efficient way of ensuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the LFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an LFENCE instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of E8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, LFENCE is encoded by any opcode of the form 0F AE Ex, where x is in the range 8-F.

### Operation

Wait\_On\_Following\_Instructions\_Until(preceding\_instructions\_complete);

### Intel C/C++ Compiler Intrinsic Equivalent

void \_mm\_lfence(void)

### Exceptions (All Modes of Operation)

#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.  
If the LOCK prefix is used.

## LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 01 /2	LGDT <i>m16&amp;32</i>	M	N.E.	Valid	Load <i>m</i> into GDTR.
0F 01 /3	LIDT <i>m16&amp;32</i>	M	N.E.	Valid	Load <i>m</i> into IDTR.
0F 01 /2	LGDT <i>m16&amp;64</i>	M	Valid	N.E.	Load <i>m</i> into GDTR.
0F 01 /3	LIDT <i>m16&amp;64</i>	M	Valid	N.E.	Load <i>m</i> into IDTR.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

In 64-bit mode, the instruction's operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

See “SGDT—Store Global Descriptor Table Register” in Chapter 4, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for information on storing the contents of the GDTR and IDTR.

## Operation

```

IF Instruction is LIDT
  THEN
    IF OperandSize = 16
      THEN
        IDTR(Limit) ← SRC[0:15];
        IDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
      ELSE IF 32-bit Operand Size
        THEN
          IDTR(Limit) ← SRC[0:15];
          IDTR(Base) ← SRC[16:47];
        FI;
      ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
        THEN
          IDTR(Limit) ← SRC[0:15];
          IDTR(Base) ← SRC[16:79];
        FI;
      FI;
    ELSE (* Instruction is LGDT *)
      IF OperandSize = 16
        THEN
          GDTR(Limit) ← SRC[0:15];
          GDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
        ELSE IF 32-bit Operand Size
          THEN
            GDTR(Limit) ← SRC[0:15];
            GDTR(Base) ← SRC[16:47];
          FI;
        ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
          THEN
            GDTR(Limit) ← SRC[0:15];
            GDTR(Base) ← SRC[16:79];
          FI;
        FI;
      FI;
    FI;
  FI;

```

## Flags Affected

None

## Protected Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If the current privilege level is not 0. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

**Real-Address Mode Exceptions**

#UD	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

#UD	If the LOCK prefix is used.
#GP	If the current privilege level is not 0.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0. If the memory address is in a non-canonical form.
#UD	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.

**LLDT—Load Local Descriptor Table Register**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /2	LLDT <i>r/m16</i>	M	Valid	Valid	Load segment selector <i>r/m16</i> into LDTR.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA	NA

**Description**

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses the segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If bits 2-15 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes silently. However, all subsequent references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. This instruction can only be executed in protected mode or 64-bit mode.

In 64-bit mode, the operand size is fixed at 16 bits.

**Operation**

```
IF SRC(Offset) > descriptor table limit
    THEN #GP(segment selector); FI;
```

```
IF segment selector is valid
```

```
    Read segment descriptor;
```

```
    IF SegmentDescriptor(Type) ≠ LDT
        THEN #GP(segment selector); FI;
```

```
    IF segment descriptor is not present
        THEN #NP(segment selector); FI;
```

```
    LDTR(SegmentSelector) ← SRC;
```

```
    LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;
```

```
ELSE LDTR ← INVALID
```

```
FI;
```

**Flags Affected**

None

**Protected Mode Exceptions**

#GP(0)	If the current privilege level is not 0. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table. Segment selector is beyond GDT limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD	The LLDT instruction is not recognized in real-address mode.
-----	--

**Virtual-8086 Mode Exceptions**

#UD	The LLDT instruction is not recognized in virtual-8086 mode.
-----	--

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0. If the memory address is in a non-canonical form.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table. Segment selector is beyond GDT limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.



## LMSW—Load Machine Status Word

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 /6	LMSW <i>r/m16</i>	M	Valid	Valid	Loads <i>r/m16</i> in machine status word of CR0.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA	NA

### Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used to clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on IA-32 and Intel 64 processors beginning with Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Note that the operand size is fixed at 16 bits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

CR0[0:3] ← SRC[0:3];

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	The LMSW instruction is not recognized in virtual-8086 mode.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0. If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

## LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F0	LOCK	Z0	Valid	Valid	Asserts LOCK# signal for duration of the accompanying instruction.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

In most IA-32 and all Intel 64 processors, locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below for more details.

The LOCK prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. If the LOCK prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception (#UD) may be generated. An undefined opcode exception will also be generated if the LOCK prefix is used with any instruction not in the above list. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism ensures that the operation is carried out atomically with regards to memory. See "Effects of a Locked Operation on Internal Processor Caches" in Chapter 8 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information on locking of caches.

### Operation

AssertLOCK#(DurationOfAccompanyingInstruction);

### Flags Affected

None

### Protected Mode Exceptions

#UD If the LOCK prefix is used with an instruction not listed: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG.

Other exceptions can be generated by the instruction when the LOCK prefix is applied.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## LODS/LODSB/LODSW/LODSD/LODSQ—Load String

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
AC	LODS <i>m8</i>	ZO	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODS <i>m16</i>	ZO	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODS <i>m32</i>	ZO	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODS <i>m64</i>	ZO	Valid	N.E.	Load qword at address (R)SI into RAX.
AC	LODSB	ZO	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODSW	ZO	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODSD	ZO	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODSQ	ZO	Valid	N.E.	Load qword at address (R)SI into RAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

### Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The DS segment may be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. LODS/LODSQ load the quadword at address (R)SI into RAX. The (R)SI register is then incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B*, for a description of the REP prefix.

## Operation

```

IF AL ← SRC; (* Byte load *)
    THEN AL ← SRC; (* Byte load *)
        IF DF = 0
            THEN (E)SI ← (E)SI + 1;
            ELSE (E)SI ← (E)SI - 1;
        FI;
ELSE IF AX ← SRC; (* Word load *)
    THEN IF DF = 0
        THEN (E)SI ← (E)SI + 2;
        ELSE (E)SI ← (E)SI - 2;
    IF;
    FI;
ELSE IF EAX ← SRC; (* Doubleword load *)
    THEN IF DF = 0
        THEN (E)SI ← (E)SI + 4;
        ELSE (E)SI ← (E)SI - 4;
    FI;
    FI;
ELSE IF RAX ← SRC; (* Quadword load *)
    THEN IF DF = 0
        THEN (R)SI ← (R)SI + 8;
        ELSE (R)SI ← (R)SI - 8;
    FI;
    FI;
FI;

```

## Flags Affected

None

## Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## LOOP/LOOP<sub>cc</sub>—Loop According to ECX Counter

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
E2 <i>cb</i>	LOOP <i>rel8</i>	D	Valid	Valid	Decrement count; jump short if count $\neq$ 0.
E1 <i>cb</i>	LOOPE <i>rel8</i>	D	Valid	Valid	Decrement count; jump short if count $\neq$ 0 and ZF = 1.
E0 <i>cb</i>	LOOPNE <i>rel8</i>	D	Valid	Valid	Decrement count; jump short if count $\neq$ 0 and ZF = 0.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA

### Description

Performs a loop operation using the RCX, ECX or CX register as a counter (depending on whether address size is 64 bits, 32 bits, or 16 bits). Note that the LOOP instruction ignores REX.W; but 64-bit address size can be over-ridden using a 67H prefix.

Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the IP/EIP/RIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of  $-128$  to  $+127$  are allowed with this instruction.

Some forms of the loop instruction (LOOP<sub>cc</sub>) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP<sub>cc</sub> instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

### Operation

```
IF (AddressSize = 32)
    THEN Count is ECX;
ELSE IF (AddressSize = 64)
    Count is RCX;
ELSE Count is CX;
FI;
```

```
Count ← Count - 1;
```

```
IF Instruction is not LOOP
    THEN
        IF (Instruction ← LOOPE) or (Instruction ← LOOPZ)
            THEN IF (ZF = 1) and (Count  $\neq$  0)
                THEN BranchCond ← 1;
                ELSE BranchCond ← 0;
            FI;
        ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
            IF (ZF = 0) and (Count  $\neq$  0)
                THEN BranchCond ← 1;
                ELSE BranchCond ← 0;
```



```

        FI;
    FI;
ELSE (* Instruction = LOOP *)
    IF (Count ≠ 0)
        THEN BranchCond ← 1;
        ELSE BranchCond ← 0;
    FI;
FI;

IF BranchCond = 1
    THEN
        IF OperandSize = 32
            THEN EIP ← EIP + SignExtend(DEST);
            ELSE IF OperandSize = 64
                THEN RIP ← RIP + SignExtend(DEST);
                FI;
            ELSE IF OperandSize = 16
                THEN EIP ← EIP AND 0000FFFFH;
                FI;
        FI;
        IF OperandSize = (32 or 64)
            THEN IF (R/E)IP < CS.Base or (R/E)IP > CS.Limit
                #GP; FI;
                FI;
        FI;
    ELSE
        Terminate loop and continue program execution at (R/E)IP;
FI;

```

**Flags Affected**

None

**Protected Mode Exceptions**

#GP(0)            If the offset being jumped to is beyond the limits of the CS segment.  
 #UD            If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP            If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.  
 #UD            If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)            If the offset being jumped to is in a non-canonical form.  
 #UD            If the LOCK prefix is used.

## LSL—Load Segment Limit

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 03 /r	LSL <i>r16</i> , <i>r16/m16</i>	RM	Valid	Valid	Load: <i>r16</i> ← segment limit, selector <i>r16/m16</i> .
OF 03 /r	LSL <i>r32</i> , <i>r32/m16</i> *	RM	Valid	Valid	Load: <i>r32</i> ← segment limit, selector <i>r32/m16</i> .
REX.W + OF 03 /r	LSL <i>r64</i> , <i>r32/m16</i> *	RM	Valid	Valid	Load: <i>r64</i> ← segment limit, selector <i>r32/m16</i> .

### NOTES:

\* For all loads (regardless of destination sizing), only bits 16-0 are used. Other bits are ignored.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg ( <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA

### Description

Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit “raw” limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

**Table 3-55. Segment and Gate Descriptor Types**

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT <sup>1</sup>	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	No	Reserved	No
5	16-bit/32-bit task gate	No	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	64-bit TSS <sup>1</sup>	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS <sup>1</sup>	Yes
C	32-bit call gate	No	64-bit call gate	No
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

**NOTES:**

1. In this case, the descriptor comprises 16 bytes; bits 12:8 of the upper 4 bytes must be 0.

**Operation**

IF SRC(Offset) > descriptor table limit  
 THEN ZF ← 0; FI;

Read segment descriptor;

IF SegmentDescriptor(Type) ≠ conforming code segment  
 and (CPL > DPL) OR (RPL > DPL)  
 or Segment type is not valid for instruction

THEN

ZF ← 0;

ELSE

temp ← SegmentLimit([SRC]);

IF (G ← 1)

THEN temp ← ShiftLeft(12, temp) OR 00000FFFH;

ELSE IF OperandSize = 32

THEN DEST ← temp; FI;

ELSE IF OperandSize = 64 (\* REX.W used \*)

THEN DEST (\* Zero-extended \*) ← temp; FI;

ELSE (\* OperandSize = 16 \*)

DEST ← temp AND FFFFH;

FI;

FI;

## Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is set to 0.

## Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#UD	The LSL instruction cannot be executed in real-address mode.
-----	--

## Virtual-8086 Mode Exceptions

#UD	The LSL instruction cannot be executed in virtual-8086 mode.
-----	--

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory operand effective address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## LTR—Load Task Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /3	LTR <i>r/m16</i>	M	Valid	Valid	Load <i>r/m16</i> into task register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA	NA

### Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

### Operation

IF SRC is a NULL selector  
THEN #GP(0);

IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global  
THEN #GP(segment selector); FI;

Read segment descriptor;

IF segment descriptor is not for an available TSS  
THEN #GP(segment selector); FI;

IF segment descriptor is not present  
THEN #NP(segment selector); FI;

TSSsegmentDescriptor(busy) ← 1;

(\* Locked read-modify-write operation on the entire descriptor when setting busy flag \*)

TaskRegister(SegmentSelector) ← SRC;

TaskRegister(SegmentDescriptor) ← TSSsegmentDescriptor;

### Flags Affected

None

**Protected Mode Exceptions**

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the source operand contains a NULL segment selector.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If the source selector points to a segment that is not a TSS or to one for a task that is already busy.</p> <p>If the selector points to LDT or is beyond the GDT limit.</p>
#NP(selector)	If the TSS is marked not present.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD	The LTR instruction is not recognized in real-address mode.
-----	---

**Virtual-8086 Mode Exceptions**

#UD	The LTR instruction is not recognized in virtual-8086 mode.
-----	---

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the memory address is in a non-canonical form.</p> <p>If the source operand contains a NULL segment selector.</p>
#GP(selector)	<p>If the source selector points to a segment that is not a TSS or to one for a task that is already busy.</p> <p>If the selector points to LDT or is beyond the GDT limit.</p> <p>If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero.</p>
#NP(selector)	If the TSS is marked not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

## LZCNT— Count the Number of Leading Zero Bits

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F BD /r	RM	V/V	LZCNT	Count the number of leading zero bits in r/m16, return result in r16.
LZCNT r16, r/m16				
F3 0F BD /r	RM	V/V	LZCNT	Count the number of leading zero bits in r/m32, return result in r32.
LZCNT r32, r/m32				
F3 REX.W 0F BD /r	RM	V/N.E.	LZCNT	Count the number of leading zero bits in r/m64, return result in r64.
LZCNT r64, r/m64				

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Counts the number of leading most significant zero bits in a source operand (second operand) returning the result into a destination (first operand).

LZCNT differs from BSR. For example, LZCNT will produce the operand size when the input operand is zero. It should be noted that on processors that do not support LZCNT, the instruction byte encoding is executed as BSR.

In 64-bit mode 64-bit operand size requires REX.W=1.

### Operation

```

temp ← OperandSize - 1
DEST ← 0
WHILE (temp >= 0) AND (Bit(SRC, temp) = 0)
DO
    temp ← temp - 1
    DEST ← DEST + 1
OD

IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI

IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI

```

### Flags Affected

ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise, CF flag is set to 1 if input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

## Intel C/C++ Compiler Intrinsic Equivalent

LZCNT:      unsigned \_\_int32 \_lzcnt\_u32(unsigned \_\_int32 src);

LZCNT:      unsigned \_\_int64 \_lzcnt\_u64(unsigned \_\_int64 src);

## Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.

## Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

## 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.



**MASKMOVDQU—Store Selected Bytes of Double Quadword**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F F7 /r MASKMOVDQU <i>xmm1</i> , <i>xmm2</i>	RM	V/V	SSE2	Selectively write bytes from <i>xmm1</i> to memory location using the byte mask in <i>xmm2</i> . The default memory location is specified by DS:DI/EDI/RDI.
VEX.128.66.0F.WIG F7 /r VMASKMOVDQU <i>xmm1</i> , <i>xmm2</i>	RM	V/V	AVX	Selectively write bytes from <i>xmm1</i> to memory location using the byte mask in <i>xmm2</i> . The default memory location is specified by DS:DI/EDI/RDI.

**Instruction Operand Encoding<sup>1</sup>**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

**Description**

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

Behavior with a mask of all 0s is as follows:

- No data will be written to memory.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVDQU instruction can be used to improve performance of algorithms that need to merge data on a byte-by-byte basis. MASKMOVDQU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

If VMASKMOVDQU is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

---

1. ModRM.MOD = 011B required

## Operation

```

IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI + 1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 14th bytes in source operand *)
IF (MASK[127] = 1)
    THEN DEST[DI/EDI + 15] ← SRC[127:120] ELSE (* Memory location unchanged *); FI;

```

## Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)
```

## Other Exceptions

See Exceptions Type 4; additionally

```

#UD                If VEX.L = 1
                   If VEX.vvvv ≠ 1111B.

```

**MASKMOVQ—Store Selected Bytes of Quadword**

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F F7 /r MASKMOVQ <i>mm1</i> , <i>mm2</i>	RM	Valid	Valid	Selectively write bytes from <i>mm1</i> to memory location using the byte mask in <i>mm2</i> . The default memory location is specified by DS:DI/EDI/RDI.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

**Description**

Stores selected bytes from the source operand (first operand) into a 64-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are MMX technology registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQ instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction causes a transition from x87 FPU to MMX technology state (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]).

The behavior of the MASKMOVQ instruction with a mask of all 0s is as follows:

- No data will be written to memory.
- Transition from x87 FPU to MMX technology state will occur.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- Signaling of breakpoints (code or data) is not guaranteed (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, the memory address is specified by DS:RDI.

## Operation

```

IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI + 1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 6th bytes in source operand *)
IF (MASK[63] = 1)
    THEN DEST[DI/EDI + 15] ← SRC[63:56] ELSE (* Memory location unchanged *); FI;

```

## Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmove_si64(__m64d, __m64n, char * p)
```

## Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## MAXPD—Maximum of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5F /r MAXPD xmm1, xmm2/m128	A	V/V	SSE2	Return the maximum double-precision floating-point values between xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 5F /r VMAXPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum double-precision floating-point values between xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256.
EVEX.NDS.128.66.0F.W1 5F /r VMAXPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 5F /r VMAXPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F	Return the maximum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPD can be emulated using a sequence of instructions, such as a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

### Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}
```

### VMAXPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← MAX(SRC1[i+63:i], SRC2[63:0])
                ELSE
                    DEST[i+63:i] ← MAX(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE DEST[i+63:i] ← 0 ; zeroing-masking
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

### VMAXPD (VEX.256 encoded version)

```
DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MAX(SRC1[127:64], SRC2[127:64])
DEST[191:128] ← MAX(SRC1[191:128], SRC2[191:128])
DEST[255:192] ← MAX(SRC1[255:192], SRC2[255:192])
DEST[MAXVL-1:256] ← 0
```

### VMAXPD (VEX.128 encoded version)

```
DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MAX(SRC1[127:64], SRC2[127:64])
DEST[MAXVL-1:128] ← 0
```

**MAXPD (128-bit Legacy SSE version)**

$\text{DEST}[63:0] \leftarrow \text{MAX}(\text{DEST}[63:0], \text{SRC}[63:0])$

$\text{DEST}[127:64] \leftarrow \text{MAX}(\text{DEST}[127:64], \text{SRC}[127:64])$

$\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

`VMAXPD __m512d _mm512_max_pd( __m512d a, __m512d b);`

`VMAXPD __m512d _mm512_mask_max_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);`

`VMAXPD __m512d _mm512_maskz_max_pd( __mmask8 k, __m512d a, __m512d b);`

`VMAXPD __m512d _mm512_max_round_pd( __m512d a, __m512d b, int);`

`VMAXPD __m512d _mm512_mask_max_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);`

`VMAXPD __m512d _mm512_maskz_max_round_pd( __mmask8 k, __m512d a, __m512d b, int);`

`VMAXPD __m256d _mm256_mask_max_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);`

`VMAXPD __m256d _mm256_maskz_max_pd( __mmask8 k, __m256d a, __m256d b);`

`VMAXPD __m128d _mm_mask_max_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);`

`VMAXPD __m128d _mm_maskz_max_pd( __mmask8 k, __m128d a, __m128d b);`

`VMAXPD __m256d _mm256_max_pd( __m256d a, __m256d b);`

`(V)VMAXPD __m128d _mm_max_pd( __m128d a, __m128d b);`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MAXPS—Maximum of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5F /r MAXPS xmm1, xmm2/m128	A	V/V	SSE	Return the maximum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 5F /r VMAXPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum single-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 5F /r VMAXPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 5F /r VMAXPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F	Return the maximum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MAXPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.



**Operation**

MAX(SRC1, SRC2)

```

{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

```

**VMAXPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[31:0])

ELSE

DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE DEST[i+31:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMAXPS (VEX.256 encoded version)**

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])

DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])

DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])

DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])

DEST[159:128] ← MAX(SRC1[159:128], SRC2[159:128])

DEST[191:160] ← MAX(SRC1[191:160], SRC2[191:160])

DEST[223:192] ← MAX(SRC1[223:192], SRC2[223:192])

DEST[255:224] ← MAX(SRC1[255:224], SRC2[255:224])

DEST[MAXVL-1:256] ← 0

**VMAXPS (VEX.128 encoded version)**

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])

DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])

DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])

DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])

DEST[MAXVL-1:128] ← 0

**MAXPS (128-bit Legacy SSE version)**

$\text{DEST}[31:0] \leftarrow \text{MAX}(\text{DEST}[31:0], \text{SRC}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{MAX}(\text{DEST}[63:32], \text{SRC}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{MAX}(\text{DEST}[95:64], \text{SRC}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{MAX}(\text{DEST}[127:96], \text{SRC}[127:96])$   
 $\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

`VMAXPS __m512 __mm512_max_ps( __m512 a, __m512 b);`  
`VMAXPS __m512 __mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);`  
`VMAXPS __m512 __mm512_maskz_max_ps( __mmask16 k, __m512 a, __m512 b);`  
`VMAXPS __m512 __mm512_max_round_ps( __m512 a, __m512 b, int);`  
`VMAXPS __m512 __mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);`  
`VMAXPS __m512 __mm512_maskz_max_round_ps( __mmask16 k, __m512 a, __m512 b, int);`  
`VMAXPS __m256 __mm256_mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);`  
`VMAXPS __m256 __mm256_maskz_max_ps( __mmask8 k, __m256 a, __m256 b);`  
`VMAXPS __m128 __mm_mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);`  
`VMAXPS __m128 __mm_maskz_max_ps( __mmask8 k, __m128 a, __m128 b);`  
`VMAXPS __m256 __mm256_max_ps ( __m256 a, __m256 b);`  
`MAXPS __m128 __mm_max_ps ( __m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

**MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD xmm1, xmm2/m64	A	V/V	SSE2	Return the maximum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.LIG.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

**Description**

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN of either source operand be returned, the action of MAXSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSD is encoded with VEX.L=0. Encoding VMAXSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

```

MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

```

**VMAXSD (EVEX encoded version)**

```

IF k1[0] or *no writemask*
    THEN DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[63:0] ← 0
        FI;
    FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**VMAXSD (VEX.128 encoded version)**

```

DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**MAXSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← MAX(DEST[63:0], SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMAXSD __m128d _mm_max_round_sd( __m128d a, __m128d b, int);
VMAXSD __m128d _mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d _mm_maskz_max_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid (Including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

**MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS xmm1, xmm2/m32	A	V/V	SSE	Return the maximum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.LIG.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

**Description**

Compares the low single-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSS is encoded with VEX.L=0. Encoding VMAXSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

```

MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 > SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

```

**VMAXSS (EVEX encoded version)**

```

IF k1[0] or *no writemask*
    THEN DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**VMAXSS (VEX.128 encoded version)**

```

DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**MAXSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← MAX(DEST[31:0], SRC[31:0])
DEST[MAXVL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMAXSS __m128 _mm_max_round_ss( __m128 a, __m128 b, int);
VMAXSS __m128 _mm_mask_max_round_ss( __m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128 _mm_maskz_max_round_ss( __mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128 _mm_max_ss( __m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

Invalid (Including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## MFENCE—Memory Fence

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF AE F0	MFENCE	Z0	Valid	Valid	Serializes load and store operations.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction.<sup>1</sup> The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F0. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, MFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 0-7.

### Operation

Wait\_On\_Following\_Loads\_And\_Stores\_Until(preceding\_loads\_and\_stores\_globally\_visible);

### Intel C/C++ Compiler Intrinsic Equivalent

void \_mm\_mfence(void)

### Exceptions (All Modes of Operation)

#UD If CPUID.01H:EDX.SSE2[bit 26] = 0.  
If the LOCK prefix is used.

1. A load instruction is considered to become globally visible when the value to be loaded into its destination register is determined.

## MINPD—Minimum of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5D /r MINPD xmm1, xmm2/m128	A	V/V	SSE2	Return the minimum double-precision floating-point values between xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 5D /r VMINPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum double-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 5D /r VMINPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 5D /r VMINPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 5D /r VMINPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F	Return the minimum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.



**Operation**

MIN(SRC1, SRC2)

```

{
  IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
  ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
  ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
  ELSE DEST ← SRC2;
  FI;
}

```

**VMINPD (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← MIN(SRC1[i+63:i], SRC2[63:0])

ELSE

DEST[i+63:i] ← MIN(SRC1[i+63:i], SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMINPD (VEX.256 encoded version)**

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])

DEST[191:128] ← MIN(SRC1[191:128], SRC2[191:128])

DEST[255:192] ← MIN(SRC1[255:192], SRC2[255:192])

**VMINPD (VEX.128 encoded version)**

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])

DEST[MAXVL-1:128] ← 0

**MINPD (128-bit Legacy SSE version)**

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])

DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMINPD __m512d __mm512_min_pd( __m512d a, __m512d b);
VMINPD __m512d __mm512_mask_min_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d __mm512_maskz_min_pd( __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d __mm512_min_round_pd( __m512d a, __m512d b, int);
VMINPD __m512d __mm512_mask_min_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m512d __mm512_maskz_min_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m256d __mm256_mask_min_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VMINPD __m256d __mm256_maskz_min_pd( __mmask8 k, __m256d a, __m256d b);
VMINPD __m128d __mm_mask_min_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMINPD __m128d __mm_maskz_min_pd( __mmask8 k, __m128d a, __m128d b);
VMINPD __m256d __mm256_min_pd ( __m256d a, __m256d b);
MINPD __m128d __mm_min_pd ( __m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MINPS—Minimum of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 5D /r MINPS xmm1, xmm2/m128	A	V/V	SSE	Return the minimum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.OF.WIG 5D /r VMINPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.OF.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.OF.W0 5D /r VMINPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.OF.W0 5D /r VMINPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.OF.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F	Return the minimum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation**

```

MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

```

**VMINPS (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE DEST[i+31:i] ← 0 ; zeroing-masking
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VMINPS (VEX.256 encoded version)**

```

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[159:128] ← MIN(SRC1[159:128], SRC2[159:128])
DEST[191:160] ← MIN(SRC1[191:160], SRC2[191:160])
DEST[223:192] ← MIN(SRC1[223:192], SRC2[223:192])
DEST[255:224] ← MIN(SRC1[255:224], SRC2[255:224])

```

**VMINPS (VEX.128 encoded version)**

```

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] ← 0

```

**MINPS (128-bit Legacy SSE version)**

$DEST[31:0] \leftarrow \text{MIN}(SRC1[31:0], SRC2[31:0])$   
 $DEST[63:32] \leftarrow \text{MIN}(SRC1[63:32], SRC2[63:32])$   
 $DEST[95:64] \leftarrow \text{MIN}(SRC1[95:64], SRC2[95:64])$   
 $DEST[127:96] \leftarrow \text{MIN}(SRC1[127:96], SRC2[127:96])$   
 $DEST[\text{MAXVL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

`VMINPS __m512 __mm512_min_ps( __m512 a, __m512 b);`  
`VMINPS __m512 __mm512_mask_min_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);`  
`VMINPS __m512 __mm512_maskz_min_ps( __mmask16 k, __m512 a, __m512 b);`  
`VMINPS __m512 __mm512_min_round_ps( __m512 a, __m512 b, int);`  
`VMINPS __m512 __mm512_mask_min_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);`  
`VMINPS __m512 __mm512_maskz_min_round_ps( __mmask16 k, __m512 a, __m512 b, int);`  
`VMINPS __m256 __mm256_mask_min_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);`  
`VMINPS __m256 __mm256_maskz_min_ps( __mmask8 k, __m256 a, __m256 b);`  
`VMINPS __m128 __mm_mask_min_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);`  
`VMINPS __m128 __mm_maskz_min_ps( __mmask8 k, __m128 a, __m128 b);`  
`VMINPS __m256 __mm256_min_ps( __m256 a, __m256 b);`  
`MINPS __m128 __mm_min_ps( __m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5D /r MINSD xmm1, xmm2/m64	A	V/V	SSE2	Return the minimum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.LIG.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5D /r VMINSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, then SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMINSD is encoded with VEX.L=0. Encoding VMINSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation**

```

MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

```

**MINSD (EVEX encoded version)**

```

IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                             ; zeroing-masking
            THEN DEST[63:0] ← 0
        FI;
    FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**MINSD (VEX.128 encoded version)**

```

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**MINSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[MAXVL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMINSD __m128d _mm_min_round_sd(__m128d a, __m128d b, int);
VMINSD __m128d _mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSD __m128d _mm_maskz_min_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MINSD __m128d _mm_min_sd(__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid (including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5D /r MINSS xmm1,xmm2/m32	A	V/V	SSE	Return the minimum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.LIG.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Compares the low single-precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSB can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by (E)VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMINSS is encoded with VEX.L=0. Encoding VMINSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.



**Operation**

MIN(SRC1, SRC2)

```

{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ← SRC2;
    ELSE IF (SRC1 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC2 = SNaN) THEN DEST ← SRC2; FI;
    ELSE IF (SRC1 < SRC2) THEN DEST ← SRC1;
    ELSE DEST ← SRC2;
    FI;
}

```

**MINSS (EVEX encoded version)**

IF k1[0] or \*no writemask\*

```

THEN    DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
ELSE
    IF *merging-masking*           ; merging-masking
    THEN *DEST[31:0] remains unchanged*
    ELSE                           ; zeroing-masking
    THEN DEST[31:0] ← 0
    FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**VMINSS (VEX.128 encoded version)**

```

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**MINSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMINSS __m128 _mm_min_round_ss( __m128 a, __m128 b, int);
VMINSS __m128 _mm_mask_min_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMINSS __m128 _mm_maskz_min_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MINSS __m128 _mm_min_ss(__m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

Invalid (Including QNaN Source Operand), Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MONITOR—Set Up Monitor Address

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 C8	MONITOR	Z0	Valid	Valid	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is DS:RAX/EAX/AX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The address is specified in RAX/EAX/AX and the size is based on the effective address size of the encoded instruction. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 8, “Multiple-Processor Management” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MONITOR may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32\_MISC\_ENABLE MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

The instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX (RAX in 64-bit mode) as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

### Intel C/C++ Compiler Intrinsic Equivalent

MONITOR: `void _mm_monitor(void const *p, unsigned extensions,unsigned hints)`

### Numeric Exceptions

None

**Protected Mode Exceptions**

#GP(0)	<p>If the value in EAX is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If ECX <math>\neq</math> 0.</p>
#SS(0)	If the value in EAX is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#UD	<p>If CPUID.01H:ECX.MONITOR[bit 3] = 0.</p> <p>If current privilege level is not 0.</p>

**Real Address Mode Exceptions**

#GP	<p>If the CS, DS, ES, FS, or GS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.</p> <p>If ECX <math>\neq</math> 0.</p>
#SS	If the SS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

**Virtual 8086 Mode Exceptions**

#UD	The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).
-----	--

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	<p>If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form.</p> <p>If RCX <math>\neq</math> 0.</p>
#SS(0)	If the SS register is used to access memory and the value in EAX is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	<p>If the current privilege level is not 0.</p> <p>If CPUID.01H:ECX.MONITOR[bit 3] = 0.</p>

**MOV—Move**

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV r/m8,r8	MR	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8 <sup>***</sup> ,r8 <sup>***</sup>	MR	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16,r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32,r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64,r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8,r/m8	RM	Valid	Valid	Move r/m8 to r8.
REX + 8A /r	MOV r8 <sup>***</sup> ,r/m8 <sup>***</sup>	RM	Valid	N.E.	Move r/m8 to r8.
8B /r	MOV r16,r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32,r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64,r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16,Sreg <sup>**</sup>	MR	Valid	Valid	Move segment register to r/m16.
REX.W + 8C /r	MOV r16/r32/m16, Sreg <sup>**</sup>	MR	Valid	Valid	Move zero extended 16-bit segment register to r16/r32/r64/m16.
REX.W + 8C /r	MOV r64/m16, Sreg <sup>**</sup>	MR	Valid	Valid	Move zero extended 16-bit segment register to r64/m16.
8E /r	MOV Sreg,r/m16 <sup>**</sup>	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg,r/m64 <sup>**</sup>	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL,moffs8 <sup>*</sup>	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL,moffs8 <sup>*</sup>	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX,moffs16 <sup>*</sup>	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32 <sup>*</sup>	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX,moffs64 <sup>*</sup>	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8,AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8 <sup>***</sup> ,AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16 <sup>*</sup> ,AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32 <sup>*</sup> ,EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64 <sup>*</sup> ,RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8, imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8 <sup>***</sup> , imm8	OI	Valid	N.E.	Move imm8 to r8.
B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64, imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /O ib	MOV r/m8, imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /O ib	MOV r/m8 <sup>***</sup> , imm8	MI	Valid	N.E.	Move imm8 to r/m8.
C7 /O iw	MOV r/m16, imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /O id	MOV r/m32, imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /O id	MOV r/m64, imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

**NOTES:**

- \* The *moffs8*, *moffs16*, *moffs32* and *moffs64* operands specify a simple offset relative to the segment base, where 8, 16, 32 and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32 or 64 bits.
- \*\* In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following “Description” section for further information).
- \*\*\*In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FD	AL/AX/EAX/RAX	Moffs	NA	NA
TD	Moffs (w)	AL/AX/EAX/RAX	NA	NA
OI	opcode + rd (w)	imm8/16/32/64	NA	NA
MI	ModRM:r/m (w)	imm8/16/32/64	NA	NA

**Description**

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the “Operation” algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A NULL segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction inhibits all interrupts until after the execution of the next instruction. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, *stack-pointer value*) before an interrupt occurs<sup>1</sup>. Be aware that the LSS instruction offers a more efficient method of loading the SS and ESP registers.

When executing MOV Reg, Sreg, the processor copies the content of Sreg to the 16 least significant bits of the general-purpose register. The upper bits of the destination register are zero for most IA-32 processors (Pentium

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a MOV SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that load the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

In the following sequence, interrupts may be recognized before MOV ESP, EBP executes:

```
MOV SS, EDX
MOV SS, EAX
MOV ESP, EBP
```

Pro processors and later) and all Intel 64 processors, with the exception that bits 31:16 are undefined for Intel Quark X1000 processors, Pentium and earlier processors.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

DEST  $\leftarrow$  SRC;

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor to which it points.

```
IF SS is loaded
  THEN
    IF segment selector is NULL
      THEN #GP(0); FI;
    IF segment selector index is outside descriptor table limits
      or segment selector's RPL  $\neq$  CPL
      or segment is not a writable data segment
      or DPL  $\neq$  CPL
      THEN #GP(selector); FI;
    IF segment not marked present
      THEN #SS(selector);
    ELSE
      SS  $\leftarrow$  segment selector;
      SS  $\leftarrow$  segment descriptor; FI;
```

FI;

```
IF DS, ES, FS, or GS is loaded with non-NULL selector
  THEN
    IF segment selector index is outside descriptor table limits
      or segment is not a data or readable code segment
      or ((segment is a data or nonconforming code segment)
      or ((RPL > DPL) and (CPL > DPL)))
      THEN #GP(selector); FI;
    IF segment not marked present
      THEN #NP(selector);
    ELSE
      SegmentRegister  $\leftarrow$  segment selector;
      SegmentRegister  $\leftarrow$  segment descriptor; FI;
```

FI;

```
IF DS, ES, FS, or GS is loaded with NULL selector
  THEN
    SegmentRegister  $\leftarrow$  segment selector;
    SegmentRegister  $\leftarrow$  segment descriptor;
```

FI;

## Flags Affected

None

**Protected Mode Exceptions**

#GP(0)	<p>If attempt is made to load SS register with NULL segment selector.</p> <p>If the destination operand is in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains a NULL segment selector.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.</p> <p>If the SS register is being loaded and the segment pointed to is a non-writable data segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	<p>If attempt is made to load the CS register.</p> <p>If the LOCK prefix is used.</p>

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	<p>If attempt is made to load the CS register.</p> <p>If the LOCK prefix is used.</p>

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	<p>If attempt is made to load the CS register.</p> <p>If the LOCK prefix is used.</p>

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	<p>If the memory address is in a non-canonical form.</p> <p>If an attempt is made to load SS register with NULL segment selector when CPL = 3.</p> <p>If an attempt is made to load SS register with NULL segment selector when CPL &lt; 3 and CPL ≠ RPL.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>If the memory access to the descriptor table is non-canonical.</p> <p>If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.</p> <p>If the SS register is being loaded and the segment pointed to is a nonwritable data segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.</p> <p>If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.</p>
#SS(0)	If the stack address is in a non-canonical form.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	<p>If attempt is made to load the CS register.</p> <p>If the LOCK prefix is used.</p>



## MOV—Move to/from Control Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 20/r MOV r32, CR0-CR7	MR	N.E.	Valid	Move control register to r32.
OF 20/r MOV r64, CR0-CR7	MR	Valid	N.E.	Move extended control register to r64.
REX.R + OF 20 /0 MOV r64, CR8	MR	Valid	N.E.	Move extended CR8 to r64. <sup>1</sup>
OF 22 /r MOV CR0-CR7, r32	RM	N.E.	Valid	Move r32 to control register.
OF 22 /r MOV CR0-CR7, r64	RM	Valid	N.E.	Move r64 to extended control register.
REX.R + OF 22 /0 MOV CR8, r64	RM	Valid	N.E.	Move r64 to extended CR8. <sup>1</sup>

### NOTE:

1. MOV CR\* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 8 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See “Control Registers” in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read. Attempts to reference CR1, CR5, CR6, CR7, and CR9–CR15 result in undefined opcode (#UD) exceptions.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4's reserved bits will cause a general protection fault. Reserved bits in CR0 and CR3 remain clear after any load of those registers; attempts to set them have no impact. On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* for details.

The following side effects are implementation-specific for the Pentium 4, Intel Xeon, and P6 processor family: when modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, including global entries. Software should not depend on this functionality in all Intel 64 or IA-32 processors.

In 64-bit mode, the instruction's default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8-R15). Use of the REX.W prefix or 66H prefix is ignored. Use of

the REX.R prefix to specify a register other than CR8 causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 4.10.4.1, “Operations that Invalidate TLBs and Paging-Structure Caches,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

## Operation

DEST ← SRC;

## Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

## Protected Mode Exceptions

- #GP(0)
  - If the current privilege level is not 0.
  - If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
  - If an attempt is made to write a 1 to any reserved bit in CR4.
  - If an attempt is made to write 1 to CR4.PCIDE.
  - If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.
- #UD
  - If the LOCK prefix is used.
  - If an attempt is made to access CR1, CR5, CR6, or CR7.

## Real-Address Mode Exceptions

- #GP
  - If an attempt is made to write a 1 to any reserved bit in CR4.
  - If an attempt is made to write 1 to CR4.PCIDE.
  - If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).
- #UD
  - If the LOCK prefix is used.
  - If an attempt is made to access CR1, CR5, CR6, or CR7.

## Virtual-8086 Mode Exceptions

- #GP(0)
  - These instructions cannot be executed in virtual-8086 mode.

## Compatibility Mode Exceptions

- #GP(0)
  - If the current privilege level is not 0.
  - If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
  - If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
  - If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1.
  - If an attempt is made to write a 1 to any reserved bit in CR3.
  - If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
- #UD
  - If the LOCK prefix is used.
  - If an attempt is made to access CR1, CR5, CR6, or CR7.

**64-Bit Mode Exceptions**

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).</p> <p>If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.</p> <p>If an attempt is made to clear CR0.PG[bit 31].</p> <p>If an attempt is made to write a 1 to any reserved bit in CR4.</p> <p>If an attempt is made to write a 1 to any reserved bit in CR8.</p> <p>If an attempt is made to write a 1 to any reserved bit in CR3.</p> <p>If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].</p>
#UD	<p>If the LOCK prefix is used.</p> <p>If an attempt is made to access CR1, CR5, CR6, or CR7.</p> <p>If the REX.R prefix is used to specify a register other than CR8.</p>

## MOV—Move to/from Debug Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 21/ <i>r</i> MOV <i>r32</i> , DR0-DR7	MR	N.E.	Valid	Move debug register to <i>r32</i> .
OF 21/ <i>r</i> MOV <i>r64</i> , DR0-DR7	MR	Valid	N.E.	Move extended debug register to <i>r64</i> .
OF 23 / <i>r</i> MOV DR0-DR7, <i>r32</i>	RM	N.E.	Valid	Move <i>r32</i> to debug register.
OF 23 / <i>r</i> MOV DR0-DR7, <i>r64</i>	RM	Valid	N.E.	Move <i>r64</i> to extended debug register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Section 17.2, “Debug Registers”, of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*, for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE flag in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the IA-32 Architecture beginning with the Pentium processor.)

At the opcode level, the *reg* field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read.

In 64-bit mode, the instruction’s default operation size is 64 bits. Use of the REX.B prefix permits access to additional registers (R8–R15). Use of the REX.W or 66H prefix is ignored. Use of the REX.R prefix causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
IF ((DE = 1) and (SRC or DEST = DR4 or DR5))
    THEN
        #UD;
    ELSE
        DEST ← SRC;
```

FI;

### Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

**Protected Mode Exceptions**

#GP(0)	If the current privilege level is not 0.
#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. If the LOCK prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

**Real-Address Mode Exceptions**

#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. If the LOCK prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

**Virtual-8086 Mode Exceptions**

#GP(0)	The debug registers cannot be loaded or read when in virtual-8086 mode.
--------	---

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	If the current privilege level is not 0. If an attempt is made to write a 1 to any of bits 63:32 in DR6. If an attempt is made to write a 1 to any of bits 63:32 in DR7.
#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5. If the LOCK prefix is used. If the REX.R prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

## MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.
66 0F 29 /r MOVAPD xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 28 /r VMOVAPD xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 29 /r VMOVAPD xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 28 /r VMOVAPD ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed double-precision floating-point values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 29 /r VMOVAPD ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed double-precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 28 /r VMOVAPD xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 28 /r VMOVAPD ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 28 /r VMOVAPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 29 /r VMOVAPD xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 29 /r VMOVAPD ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 29 /r VMOVAPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

## Description

Moves 2, 4 or 8 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit versions), 32-byte (256-bit version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX encoded versions, the operand must be aligned to the size of the memory operand. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float64 memory location, to store the contents of a ZMM register into a 512-bit float64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit versions:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register destination are zeroed.

## Operation

### VMOVAPD (EVEX encoded versions, register-copy form)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← SRC[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE DEST[i+63:i] ← 0 ; zeroing-masking

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVAPD (EVEX encoded versions, store-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE

ELSE \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVAPD (EVEX encoded versions, load-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVAPD (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAXVL-1:256] ← 0

**VMOVAPD (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVAPD (VEX.128 encoded version, load - and register copy)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] ← 0

**MOVAPD (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)

**(V)MOVAPD (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMOVAPD __m512d _mm512_load_pd( void * m);
VMOVAPD __m512d _mm512_mask_load_pd(__m512d s, __mmask8 k, void * m);
VMOVAPD __m512d _mm512_maskz_load_pd( __mmask8 k, void * m);
VMOVAPD void _mm512_store_pd( void * d, __m512d a);
VMOVAPD void _mm512_mask_store_pd( void * d, __mmask8 k, __m512d a);
VMOVAPD __m256d _mm256_mask_load_pd(__m256d s, __mmask8 k, void * m);
VMOVAPD __m256d _mm256_maskz_load_pd( __mmask8 k, void * m);
VMOVAPD void _mm256_mask_store_pd( void * d, __mmask8 k, __m256d a);
VMOVAPD __m128d _mm_mask_load_pd(__m128d s, __mmask8 k, void * m);
VMOVAPD __m128d _mm_maskz_load_pd( __mmask8 k, void * m);
VMOVAPD void _mm_mask_store_pd( void * d, __mmask8 k, __m128d a);
MOVAPD __m256d _mm256_load_pd (double * p);
MOVAPD void _mm256_store_pd(double * p, __m256d a);
MOVAPD __m128d _mm_load_pd (double * p);
MOVAPD void _mm_store_pd(double * p, __m128d a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1.

#UD                      If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

## MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 28 /r MOVAPS xmm1, xmm2/m128	A	V/V	SSE	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
NP 0F 29 /r MOVAPS xmm2/m128, xmm1	B	V/V	SSE	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.128.0F.WIG 28 /r VMOVAPS xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
VEX.128.0F.WIG 29 /r VMOVAPS xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.256.0F.WIG 28 /r VMOVAPS ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed single-precision floating-point values from ymm2/mem to ymm1.
VEX.256.0F.WIG 29 /r VMOVAPS ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.0F.W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Moves 4, 8 or 16 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into a float32 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

VEX.256 and EVEX.256 encoded version:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

## Operation

### VMOVAPS (EVEX encoded versions, register-copy form)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+31:i] ← SRC[i+31:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+31:i] remains unchanged\*

        ELSE DEST[i+31:i] ← 0 ; zeroing-masking

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

### VMOVAPS (EVEX encoded versions, store-form)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+31:i] ←

            SRC[i+31:i]

        ELSE \*DEST[i+31:i] remains unchanged\* ; merging-masking

    FI;

ENDFOR;

**VMOVAPS (EVEX encoded versions, load-form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← SRC[i+31:i]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE DEST[i+31:i] ← 0 ; zeroing-masking

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVAPS (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAXVL-1:256] ← 0

**VMOVAPS (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVAPS (VEX.128 encoded version, load - and register copy)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] ← 0

**MOVAPS (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)

**(V)MOVAPS (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVAPS \_\_m512 \_\_mm512\_load\_ps( void \* m);

VMOVAPS \_\_m512 \_\_mm512\_mask\_load\_ps(\_\_m512 s, \_\_mmask16 k, void \* m);

VMOVAPS \_\_m512 \_\_mm512\_maskz\_load\_ps( \_\_mmask16 k, void \* m);

VMOVAPS void \_\_mm512\_store\_ps( void \* d, \_\_m512 a);

VMOVAPS void \_\_mm512\_mask\_store\_ps( void \* d, \_\_mmask16 k, \_\_m512 a);

VMOVAPS \_\_m256 \_\_mm256\_mask\_load\_ps(\_\_m256 a, \_\_mmask8 k, void \* s);

VMOVAPS \_\_m256 \_\_mm256\_maskz\_load\_ps( \_\_mmask8 k, void \* s);

VMOVAPS void \_\_mm256\_mask\_store\_ps( void \* d, \_\_mmask8 k, \_\_m256 a);

VMOVAPS \_\_m128 \_\_mm\_mask\_load\_ps(\_\_m128 a, \_\_mmask8 k, void \* s);

VMOVAPS \_\_m128 \_\_mm\_maskz\_load\_ps( \_\_mmask8 k, void \* s);

VMOVAPS void \_\_mm\_mask\_store\_ps( void \* d, \_\_mmask8 k, \_\_m128 a);

MOVAPS \_\_m256 \_\_mm256\_load\_ps( float \* p);

MOVAPS void \_\_mm256\_store\_ps(float \* p, \_\_m256 a);

MOVAPS \_\_m128 \_\_mm\_load\_ps( float \* p);

MOVAPS void \_\_mm\_store\_ps(float \* p, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 1.SSE; additionally  
#UD If VEX.vvvv != 1111B.  
EVEX-encoded instruction, see Exceptions Type E1.

## MOVBE—Move Data After Swapping Bytes

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 38 F0 /r	MOVBE <i>r16</i> , <i>m16</i>	RM	Valid	Valid	Reverse byte order in <i>m16</i> and move to <i>r16</i> .
OF 38 F0 /r	MOVBE <i>r32</i> , <i>m32</i>	RM	Valid	Valid	Reverse byte order in <i>m32</i> and move to <i>r32</i> .
REX.W + OF 38 F0 /r	MOVBE <i>r64</i> , <i>m64</i>	RM	Valid	N.E.	Reverse byte order in <i>m64</i> and move to <i>r64</i> .
OF 38 F1 /r	MOVBE <i>m16</i> , <i>r16</i>	MR	Valid	Valid	Reverse byte order in <i>r16</i> and move to <i>m16</i> .
OF 38 F1 /r	MOVBE <i>m32</i> , <i>r32</i>	MR	Valid	Valid	Reverse byte order in <i>r32</i> and move to <i>m32</i> .
REX.W + OF 38 F1 /r	MOVBE <i>m64</i> , <i>r64</i>	MR	Valid	N.E.	Reverse byte order in <i>r64</i> and move to <i>m64</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Performs a byte swap operation on the data copied from the second operand (source operand) and store the result in the first operand (destination operand). The source operand can be a general-purpose register, or memory location; the destination register can be a general-purpose register, or a memory location; however, both operands can not be registers, and only one operand can be a memory location. Both operands must be the same size, which can be a word, a doubleword or quadword.

The MOVBE instruction is provided for swapping the bytes on a read from memory or on a write to memory; thus providing support for converting little-endian values to big-endian format and vice versa.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

TEMP ← SRC

IF ( OperandSize = 16)

THEN

DEST[7:0] ← TEMP[15:8];

DEST[15:8] ← TEMP[7:0];

ELSE IF ( OperandSize = 32)

DEST[7:0] ← TEMP[31:24];

DEST[15:8] ← TEMP[23:16];

DEST[23:16] ← TEMP[15:8];

DEST[31:23] ← TEMP[7:0];

ELSE IF ( OperandSize = 64)

DEST[7:0] ← TEMP[63:56];

DEST[15:8] ← TEMP[55:48];

DEST[23:16] ← TEMP[47:40];

DEST[31:24] ← TEMP[39:32];

DEST[39:32] ← TEMP[31:24];

DEST[47:40] ← TEMP[23:16];

DEST[55:48] ← TEMP[15:8];

DEST[63:56] ← TEMP[7:0];

FI;

**Flags Affected**

None

**Protected Mode Exceptions**

#GP(0)	If the destination operand is in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used. If REPNE (F2H) prefix is used and CPUID.01H:ECX.SSE4_2[bit 20] = 0.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0. If the LOCK prefix is used. If REP (F3H) prefix is used.

## MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6E /r MOVD <i>mm</i> , <i>r/m32</i>	A	V/V	MMX	Move doubleword from <i>r/m32</i> to <i>mm</i> .
NP REX.W + 0F 6E /r MOVQ <i>mm</i> , <i>r/m64</i>	A	V/N.E.	MMX	Move quadword from <i>r/m64</i> to <i>mm</i> .
NP 0F 7E /r MOVD <i>r/m32</i> , <i>mm</i>	B	V/V	MMX	Move doubleword from <i>mm</i> to <i>r/m32</i> .
NP REX.W + 0F 7E /r MOVQ <i>r/m64</i> , <i>mm</i>	B	V/N.E.	MMX	Move quadword from <i>mm</i> to <i>r/m64</i> .
66 0F 6E /r MOVD <i>xmm</i> , <i>r/m32</i>	A	V/V	SSE2	Move doubleword from <i>r/m32</i> to <i>xmm</i> .
66 REX.W 0F 6E /r MOVQ <i>xmm</i> , <i>r/m64</i>	A	V/N.E.	SSE2	Move quadword from <i>r/m64</i> to <i>xmm</i> .
66 0F 7E /r MOVD <i>r/m32</i> , <i>xmm</i>	B	V/V	SSE2	Move doubleword from <i>xmm</i> register to <i>r/m32</i> .
66 REX.W 0F 7E /r MOVQ <i>r/m64</i> , <i>xmm</i>	B	V/N.E.	SSE2	Move quadword from <i>xmm</i> register to <i>r/m64</i> .
VEX.128.66.0F.W0 6E / VMOVD <i>xmm1</i> , <i>r32/m32</i>	A	V/V	AVX	Move doubleword from <i>r/m32</i> to <i>xmm1</i> .
VEX.128.66.0F.W1 6E /r VMOVQ <i>xmm1</i> , <i>r64/m64</i>	A	V/N.E. <sup>1</sup>	AVX	Move quadword from <i>r/m64</i> to <i>xmm1</i> .
VEX.128.66.0F.W0 7E /r VMOVD <i>r32/m32</i> , <i>xmm1</i>	B	V/V	AVX	Move doubleword from <i>xmm1</i> register to <i>r/m32</i> .
VEX.128.66.0F.W1 7E /r VMOVQ <i>r64/m64</i> , <i>xmm1</i>	B	V/N.E. <sup>1</sup>	AVX	Move quadword from <i>xmm1</i> register to <i>r/m64</i> .
EVEX.128.66.0F.W0 6E /r VMOVD <i>xmm1</i> , <i>r32/m32</i>	C	V/V	AVX512F	Move doubleword from <i>r/m32</i> to <i>xmm1</i> .
EVEX.128.66.0F.W1 6E /r VMOVQ <i>xmm1</i> , <i>r64/m64</i>	C	V/N.E. <sup>1</sup>	AVX512F	Move quadword from <i>r/m64</i> to <i>xmm1</i> .
EVEX.128.66.0F.W0 7E /r VMOVD <i>r32/m32</i> , <i>xmm1</i>	D	V/V	AVX512F	Move doubleword from <i>xmm1</i> register to <i>r/m32</i> .
EVEX.128.66.0F.W1 7E /r VMOVQ <i>r64/m64</i> , <i>xmm1</i>	D	V/N.E. <sup>1</sup>	AVX512F	Move quadword from <i>xmm1</i> register to <i>r/m64</i> .

### NOTES:

1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.



## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

## Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

MOVD/Q with XMM destination:

Moves a dword/qword integer from the source operand and stores it in the low 32/64-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32/64-bit register or 32/64-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Qword operation requires the use of REX.W=1.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of VEX.W=1.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of EVEX.W=1.

MOVD/Q with 32/64 reg/mem destination:

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VMOVD or VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

## Operation

**MOVD (when destination operand is MMX technology register)**

```
DEST[31:0] ← SRC;
DEST[63:32] ← 00000000H;
```

**MOVD (when destination operand is XMM register)**

```
DEST[31:0] ← SRC;
DEST[127:32] ← 000000000000000000000000H;
DEST[MAXVL-1:128] (Unmodified)
```

**MOVD (when source operand is MMX technology or XMM register)**

$$\text{DEST} \leftarrow \text{SRC}[31:0];$$
**VMOVD (VEX-encoded version when destination is an XMM register)**

$$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$$

$$\text{DEST}[\text{MAXVL}-1:32] \leftarrow 0$$
**MOVQ (when destination operand is XMM register)**

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0];$$

$$\text{DEST}[127:64] \leftarrow 0000000000000000\text{H};$$

$$\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}$$
**MOVQ (when destination operand is r/m64)**

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0];$$
**MOVQ (when source operand is XMM register or r/m64)**

$$\text{DEST} \leftarrow \text{SRC}[63:0];$$
**VMOVQ (VEX-encoded version when destination is an XMM register)**

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$

$$\text{DEST}[\text{MAXVL}-1:64] \leftarrow 0$$
**VMOVD (EVEX-encoded version when destination is an XMM register)**

$$\text{DEST}[31:0] \leftarrow \text{SRC}[31:0]$$

$$\text{DEST}[\text{MAXVL}-1:32] \leftarrow 0$$
**VMOVQ (EVEX-encoded version when destination is an XMM register)**

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$

$$\text{DEST}[\text{MAXVL}-1:64] \leftarrow 0$$
**Intel C/C++ Compiler Intrinsic Equivalent**

MOVD: `__m64 _mm_cvtsi32_si64 (int i)`

MOVD: `int _mm_cvtsi64_si32 ( __m64m )`

MOVD: `__m128i _mm_cvtsi32_si128 (int a)`

MOVD: `int _mm_cvtsi128_si32 ( __m128i a)`

MOVQ: `__int64 _mm_cvtsi128_si64(__m128i);`

MOVQ: `__m128i _mm_cvtsi64_si128(__int64);`

VMOVD `__m128i _mm_cvtsi32_si128( int);`

VMOVD `int _mm_cvtsi128_si32( __m128i );`

VMOVQ `__m128i _mm_cvtsi64_si128 ( __int64);`

VMOVQ `__int64 _mm_cvtsi128_si64(__m128i );`

VMOVQ `__m128i _mm_load_epi64( __m128i * s);`

VMOVQ `void _mm_store_epi64( __m128i * d, __m128i s);`

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD                      If VEX.L = 1.  
                            If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## MOVDDUP—Replicate Double FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP xmm1, xmm2/m64	A	V/V	SSE3	Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64	A	V/V	AVX	Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index double-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F2.0F.W1 12 /r VMOVDDUP xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Move double-precision floating-point value from xmm2/m64 and duplicate each element into xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 12 /r VMOVDDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512F	Move even index double-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 subject to writemask k1.
EVEX.512.F2.0F.W1 12 /r VMOVDDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F	Move even index double-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	MOVDDUP	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

For 256-bit or higher versions: Duplicates even-indexed double-precision floating-point values from the source operand (the second operand) and into adjacent pair and store to the destination operand (the first operand).

For 128-bit versions: Duplicates the low double-precision floating-point value from the source operand (the second operand) and store to the destination operand (the first operand).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register are unchanged. The source operand is XMM register or a 64-bit memory location.

VEX.128 and EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. The source operand is XMM register or a 64-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

VEX.256 and EVEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed. The source operand is YMM register or a 256-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

EVEX.512 encoded version: The destination is updated according to the writemask. The source operand is ZMM register or a 512-bit memory location.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

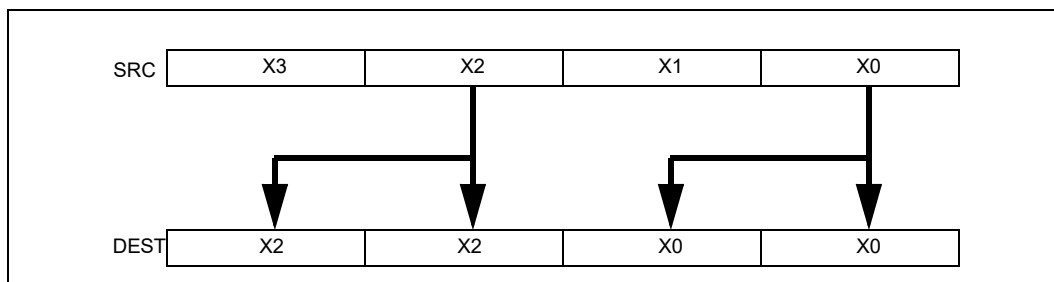


Figure 4-2. VMOVDDUP Operation

**Operation****VMOVDDUP (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

TMP\_SRC[63:0] ← SRC[63:0]

TMP\_SRC[127:64] ← SRC[63:0]

IF VL ≥ 256

    TMP\_SRC[191:128] ← SRC[191:128]

    TMP\_SRC[255:192] ← SRC[191:128]

FI;

IF VL ≥ 512

    TMP\_SRC[319:256] ← SRC[319:256]

    TMP\_SRC[383:320] ← SRC[319:256]

    TMP\_SRC[477:384] ← SRC[477:384]

    TMP\_SRC[511:484] ← SRC[477:384]

FI;

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← TMP\_SRC[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE ; zeroing-masking

            DEST[i+63:i] ← 0 ; zeroing-masking

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDDUP (VEX.256 encoded version)**

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← SRC[63:0]

DEST[191:128] ← SRC[191:128]

DEST[255:192] ← SRC[191:128]

DEST[MAXVL-1:256] ← 0

**VMOVDDUP (VEX.128 encoded version)**

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← SRC[63:0]

DEST[MAXVL-1:128] ← 0

**MOVDDUP (128-bit Legacy SSE version)**

DEST[63:0] ← SRC[63:0]

DEST[127:64] ← SRC[63:0]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVDDUP \_\_m512d \_\_mm512\_movedup\_pd(\_\_m512d a);

VMOVDDUP \_\_m512d \_\_mm512\_mask\_movedup\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VMOVDDUP \_\_m512d \_\_mm512\_maskz\_movedup\_pd(\_\_mmask8 k, \_\_m512d a);

VMOVDDUP \_\_m256d \_\_mm256\_mask\_movedup\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a);

VMOVDDUP \_\_m256d \_\_mm256\_maskz\_movedup\_pd(\_\_mmask8 k, \_\_m256d a);

VMOVDDUP \_\_m128d \_\_mm\_mask\_movedup\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VMOVDDUP \_\_m128d \_\_mm\_maskz\_movedup\_pd(\_\_mmask8 k, \_\_m128d a);

MOVDDUP \_\_m256d \_\_mm256\_movedup\_pd(\_\_m256d a);

MOVDDUP \_\_m128d \_\_mm\_movedup\_pd(\_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E5NF.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

**MOVDQA, VMOVDQA32/64—Move Aligned Packed Integer Values**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1.
66 0F 7F /r MOVDQA xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W0 6F /r VMOVDQA32 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W0 6F /r VMOVDQA32 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W0 6F /r VMOVDQA32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W0 7F /r VMOVDQA32 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W0 7F /r VMOVDQA32 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W0 7F /r VMOVDQA32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.66.0F.W1 6F /r VMOVDQA64 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 6F /r VMOVDQA64 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 6F /r VMOVDQA64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 7F /r VMOVDQA64 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 7F /r VMOVDQA64 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 7F /r VMOVDQA64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

## Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16 (EVEX.128)/32 (EVEX.256)/64 (EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.



**Operation****VMOVDQA32 (EVEX encoded versions, register-copy form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE DEST[i+31:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQA32 (EVEX encoded versions, store-form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[i+31:i]

ELSE \*DEST[i+31:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVDQA32 (EVEX encoded versions, load-form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE DEST[i+31:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQA64 (EVEX encoded versions, register-copy form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE DEST[i+63:i] ← 0                ; zeroing-masking
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VMOVDQA64 (EVEX encoded versions, store-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR;

```

**VMOVDQA64 (EVEX encoded versions, load-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE DEST[i+63:i] ← 0                ; zeroing-masking
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VMOVDQA (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAXVL-1:256] ← 0

**VMOVDQA (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVDQA (VEX.128 encoded version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] ← 0

**VMOVDQA (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)

**(V)MOVDQA (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMOVDQA32 __m512i _mm512_load_epi32( void * sa);
VMOVDQA32 __m512i _mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQA32 __m512i _mm512_maskz_load_epi32( __mmask16 k, void * sa);
VMOVDQA32 void _mm512_store_epi32(void * d, __m512i a);
VMOVDQA32 void _mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQA32 __m256i _mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQA32 __m256i _mm256_maskz_load_epi32( __mmask8 k, void * sa);
VMOVDQA32 void _mm256_store_epi32(void * d, __m256i a);
VMOVDQA32 void _mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQA32 __m128i _mm_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQA32 __m128i _mm_maskz_load_epi32( __mmask8 k, void * sa);
VMOVDQA32 void _mm_store_epi32(void * d, __m128i a);
VMOVDQA32 void _mm_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQA64 __m512i _mm512_load_epi64( void * sa);
VMOVDQA64 __m512i _mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQA64 __m512i _mm512_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void _mm512_store_epi64(void * d, __m512i a);
VMOVDQA64 void _mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQA64 __m256i _mm256_mask_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQA64 __m256i _mm256_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void _mm256_store_epi64(void * d, __m256i a);
VMOVDQA64 void _mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQA64 __m128i _mm_mask_load_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQA64 __m128i _mm_maskz_load_epi64( __mmask8 k, void * sa);
VMOVDQA64 void _mm_store_epi64(void * d, __m128i a);
VMOVDQA64 void _mm_mask_store_epi64(void * d, __mmask8 k, __m128i a);
MOVDQA void __m256i _mm256_load_si256 (__m256i * p);
MOVDQA _mm256_store_si256(_m256i *p, __m256i a);
MOVDQA __m128i _mm_load_si128 (__m128i * p);
MOVDQA void _mm_store_si128(__m128i *p, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

**MOVDQU, VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed integer values from xmm2/m128 to xmm1.
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed integer values from xmm2/m128 to xmm1.
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed integer values from ymm2/m256 to ymm1.
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/m256.
EVEX.128.F2.0F.W0 6F /r VMOVDQU8 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W0 6F /r VMOVDQU8 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W0 6F /r VMOVDQU8 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW	Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W0 7F /r VMOVDQU8 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W0 7F /r VMOVDQU8 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W0 7F /r VMOVDQU8 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW	Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F2.0F.W1 6F /r VMOVDQU16 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W1 6F /r VMOVDQU16 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W1 6F /r VMOVDQU16 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW	Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W1 7F /r VMOVDQU16 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W1 7F /r VMOVDQU16 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W1 7F /r VMOVDQU16 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW	Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W0 6F /r VMOVDQU32 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.F3.0F.W0 6F /r VMOVDQU32 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W0 6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W0 7F /r VMOVDQU32 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W0 7F /r VMOVDQU32 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W0 7F /r VMOVDQU32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W1 6F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1	C	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W1 6F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1	C	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1	C	V/V	AVX512F	Move unaligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W1 7F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W1 7F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

### EVEX encoded versions:

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) granularity according to the writemask.

#### VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Bits (MAXVL-1:256) of the destination register are zeroed.

#### 128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

**VEX.128 encoded version:** Bits (MAXVL-1:128) of the destination register are zeroed.

### Operation

#### VMOVDQU8 (EVEX encoded versions, register-copy form)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

    i ← j \* 8

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+7:i] ← SRC[i+7:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+7:i] remains unchanged\*

        ELSE DEST[i+7:i] ← 0 ; zeroing-masking

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

#### VMOVDQU8 (EVEX encoded versions, store-form)

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

    i ← j \* 8

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+7:i] ←

            SRC[i+7:i]

        ELSE \*DEST[i+7:i] remains unchanged\* ; merging-masking

    FI;

ENDFOR;

**VMOVDQU8 (EVEX encoded versions, load-form)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← SRC[i+7:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE DEST[i+7:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQU16 (EVEX encoded versions, register-copy form)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← SRC[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE DEST[i+15:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQU16 (EVEX encoded versions, store-form)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ←

SRC[i+15:i]

ELSE \*DEST[i+15:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVDQU16 (EVEX encoded versions, load-form)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[i+15:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE DEST[i+15:i] ← 0 ; zeroing-masking
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VMOVDQU32 (EVEX encoded versions, register-copy form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE DEST[i+31:i] ← 0 ; zeroing-masking
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VMOVDQU32 (EVEX encoded versions, store-form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
    FI;
ENDFOR;

```



**VMOVDQU32 (EVEX encoded versions, load-form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE DEST[i+31:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQU64 (EVEX encoded versions, register-copy form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQU64 (EVEX encoded versions, store-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;

**VMOVDQU64 (EVEX encoded versions, load-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ← SRC[i+63:i]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE DEST[i+63:i] ← 0 ; zeroing-masking

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVDQU (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAXVL-1:256] ← 0

**VMOVDQU (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

VMOVDQU (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] ← 0

**VMOVDQU (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)

**(V)MOPVDQU (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVDQU16 \_\_m512i \_\_mm512\_mask\_loadu\_epi16(\_\_m512i s, \_\_mmask32 k, void \* sa);

VMOVDQU16 \_\_m512i \_\_mm512\_maskz\_loadu\_epi16( \_\_mmask32 k, void \* sa);

VMOVDQU16 void \_\_mm512\_mask\_storeu\_epi16(void \* d, \_\_mmask32 k, \_\_m512i a);

VMOVDQU16 \_\_m256i \_\_mm256\_mask\_loadu\_epi16(\_\_m256i s, \_\_mmask16 k, void \* sa);

VMOVDQU16 \_\_m256i \_\_mm256\_maskz\_loadu\_epi16( \_\_mmask16 k, void \* sa);

VMOVDQU16 void \_\_mm256\_mask\_storeu\_epi16(void \* d, \_\_mmask16 k, \_\_m256i a);

VMOVDQU16 void \_\_mm256\_maskz\_storeu\_epi16(void \* d, \_\_mmask16 k);

VMOVDQU16 \_\_m128i \_\_mm\_mask\_loadu\_epi16(\_\_m128i s, \_\_mmask8 k, void \* sa);

VMOVDQU16 \_\_m128i \_\_mm\_maskz\_loadu\_epi16( \_\_mmask8 k, void \* sa);

VMOVDQU16 void \_\_mm\_mask\_storeu\_epi16(void \* d, \_\_mmask8 k, \_\_m128i a);

VMOVDQU32 \_\_m512i \_\_mm512\_loadu\_epi32( void \* sa);

VMOVDQU32 \_\_m512i \_\_mm512\_mask\_loadu\_epi32(\_\_m512i s, \_\_mmask16 k, void \* sa);

VMOVDQU32 \_\_m512i \_\_mm512\_maskz\_loadu\_epi32( \_\_mmask16 k, void \* sa);

VMOVDQU32 void \_\_mm512\_storeu\_epi32(void \* d, \_\_m512i a);

VMOVDQU32 void \_\_mm512\_mask\_storeu\_epi32(void \* d, \_\_mmask16 k, \_\_m512i a);

VMOVDQU32 \_\_m256i \_\_mm256\_mask\_loadu\_epi32(\_\_m256i s, \_\_mmask8 k, void \* sa);

VMOVDQU32 \_\_m256i \_\_mm256\_maskz\_loadu\_epi32( \_\_mmask8 k, void \* sa);

VMOVDQU32 void \_\_mm256\_storeu\_epi32(void \* d, \_\_m256i a);

VMOVDQU32 void \_\_mm256\_mask\_storeu\_epi32(void \* d, \_\_mmask8 k, \_\_m256i a);

VMOVDQU32 \_\_m128i \_\_mm\_mask\_loadu\_epi32(\_\_m128i s, \_\_mmask8 k, void \* sa);

```

VMOVDQU32 __m128i _mm_mask_loadu_epi32( __mmask8 k, void * sa);
VMOVDQU32 void _mm_storeu_epi32(void * d, __m128i a);
VMOVDQU32 void _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQU64 __m512i _mm512_loadu_epi64( void * sa);
VMOVDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm512_storeu_epi64(void * d, __m512i a);
VMOVDQU64 void _mm512_mask_storeu_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQU64 __m256i _mm256_mask_loadu_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQU64 __m256i _mm256_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm256_storeu_epi64(void * d, __m256i a);
VMOVDQU64 void _mm256_mask_storeu_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQU64 __m128i _mm_mask_loadu_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQU64 __m128i _mm_maskz_loadu_epi64( __mmask8 k, void * sa);
VMOVDQU64 void _mm_storeu_epi64(void * d, __m128i a);
VMOVDQU64 void _mm_mask_storeu_epi64(void * d, __mmask8 k, __m128i a);
VMOVDQU8 __m512i _mm512_mask_loadu_epi8(__m512i s, __mmask64 k, void * sa);
VMOVDQU8 __m512i _mm512_maskz_loadu_epi8( __mmask64 k, void * sa);
VMOVDQU8 void _mm512_mask_storeu_epi8(void * d, __mmask64 k, __m512i a);
VMOVDQU8 __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVDQU8 __m256i _mm256_maskz_loadu_epi8( __mmask32 k, void * sa);
VMOVDQU8 void _mm256_mask_storeu_epi8(void * d, __mmask32 k, __m256i a);
VMOVDQU8 void _mm256_maskz_storeu_epi8(void * d, __mmask32 k);
VMOVDQU8 __m128i _mm_mask_loadu_epi8(__m128i s, __mmask16 k, void * sa);
VMOVDQU8 __m128i _mm_maskz_loadu_epi8( __mmask16 k, void * sa);
VMOVDQU8 void _mm_mask_storeu_epi8(void * d, __mmask16 k, __m128i a);
MOVDQU __m256i _mm256_loadu_si256 (__m256i * p);
MOVDQU __mm256_storeu_si256(__m256i *p, __m256i a);
MOVDQU __m128i _mm_loadu_si128 (__m128i * p);
MOVDQU __mm_storeu_si128(__m128i *p, __m128i a);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD                      If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

## MOVDQ2Q—Move Quadword from XMM to MMX Technology Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F2 0F D6 /r	MOVDQ2Q <i>mm, xmm</i>	RM	Valid	Valid	Move low quadword from <i>xmm</i> to <i>mmx</i> register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Moves the low quadword from the source operand (second operand) to the destination operand (first operand). The source operand is an XMM register and the destination operand is an MMX technology register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVDQ2Q instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST ← SRC[63:0];

### Intel C/C++ Compiler Intrinsic Equivalent

MOVDQ2Q: `__m64 _mm_movepi64_pi64 ( __m128i a)`

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM If CR0.TS[bit 3] = 1.  
 #UD If CR0.EM[bit 2] = 1.  
 If CR4.OSFXSR[bit 9] = 0.  
 If CPUID.01H:EDX.SSE2[bit 26] = 0.  
 If the LOCK prefix is used.  
 #MF If there is a pending x87 FPU exception.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

**MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 12 /r MOVHLPS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm2 to low quadword of xmm1.
VEX.NDS.128.OF.WIG 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.
EVEX.NDS.128.OF.W0 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

**Instruction Operand Encoding<sup>1</sup>**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

**Description**

This instruction cannot be used for memory to register moves.

**128-bit two-argument form:**

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

**128-bit and EVEX three-argument form**

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVHLPS is encoded with VEX.L or EVEX.L'L = 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L = 1 will cause an #UD exception.

**Operation****MOVHLPS (128-bit two-argument form)**

DEST[63:0] ← SRC[127:64]

DEST[MAXVL-1:64] (Unmodified)

**VMOVHLPS (128-bit three-argument form - VEX & EVEX)**

DEST[63:0] ← SRC2[127:64]

DEST[127:64] ← SRC1[127:64]

DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVHLPS \_\_m128 \_\_mm\_movehl\_ps(\_\_m128 a, \_\_m128 b)

**SIMD Floating-Point Exceptions**

None

1. ModRM.MOD = 011B required

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally  
#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

**MOVHPD—Move High Packed Double-Precision Floating-Point Value**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD xmm1, m64	A	V/V	SSE2	Move double-precision floating-point value from m64 to high quadword of xmm1.
VEX.NDS.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	B	V/V	AVX	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
EVEX.NDS.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64	D	V/V	AVX512F	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
66 0F 17 /r MOVHPD m64, xmm1	C	V/V	SSE2	Move double-precision floating-point value from high quadword of xmm1 to m64.
VEX.128.66.0F.WIG 17 /r VMOVHPD m64, xmm1	C	V/V	AVX	Move double-precision floating-point value from high quadword of xmm1 to m64.
EVEX.128.66.0F.W1 17 /r VMOVHPD m64, xmm1	E	V/V	AVX512F	Move double-precision floating-point value from high quadword of xmm1 to m64.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

**Description**

This instruction cannot be used for register to register or memory to memory moves.

**128-bit Legacy SSE load:**

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64-bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

**VEX.128 & EVEX encoded load:**

Loads a double-precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

**128-bit store:**

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

**Operation****MOVHPD (128-bit Legacy SSE load)**

DEST[63:0] (Unmodified)

DEST[127:64] ← SRC[63:0]

DEST[MAXVL-1:128] (Unmodified)

**VMOVHPD (VEX.128 & EVEX encoded load)**

DEST[63:0] ← SRC1[63:0]

DEST[127:64] ← SRC2[63:0]

DEST[MAXVL-1:128] ← 0

**VMOVHPD (store)**

DEST[63:0] ← SRC[127:64]

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVHPD \_\_m128d \_mm\_loadh\_pd ( \_\_m128d a, double \*p)

MOVHPD void \_mm\_storeh\_pd (double \*p, \_\_m128d a)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.



## MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVHPS xmm1, m64	A	V/V	SSE	Move two packed single-precision floating-point values from m64 to high quadword of xmm1.
VEX.NDS.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
EVEX.NDS.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64	D	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
NP 0F 17 /r MOVHPS m64, xmm1	C	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1	C	V/V	AVX	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1	E	V/V	AVX512F	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

**Operation****MOVHPS (128-bit Legacy SSE load)**

DEST[63:0] (Unmodified)

DEST[127:64] ← SRC[63:0]

DEST[MAXVL-1:128] (Unmodified)

**VMOVHPS (VEX.128 and EVEX encoded load)**

DEST[63:0] ← SRC1[63:0]

DEST[127:64] ← SRC2[63:0]

DEST[MAXVL-1:128] ← 0

**VMOVHPS (store)**

DEST[63:0] ← SRC[127:64]

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVHPS \_\_m128 \_mm\_loadh\_pi ( \_\_m128 a, \_\_m64 \*p)

MOVHPS void \_mm\_storeh\_pi ( \_\_m64 \*p, \_\_m128 a)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 16 /r MOVLHPS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm2 to high quadword of xmm1.
VEX.NDS.128.OF.WIG 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.
EVEX.NDS.128.OF.W0 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

### Instruction Operand Encoding<sup>1</sup>

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction cannot be used for memory to register moves.

#### 128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register are unmodified.

#### 128-bit three-argument forms:

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L'L = 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L = 1 will cause an #UD exception.

### Operation

#### MOVLHPS (128-bit two-argument form)

DEST[63:0] (Unmodified)  
 DEST[127:64] ← SRC[63:0]  
 DEST[MAXVL-1:128] (Unmodified)

#### VMOVLHPS (128-bit three-argument form - VEX & EVEX)

DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]  
 DEST[MAXVL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS \_\_m128 \_\_mm\_movelh\_ps(\_\_m128 a, \_\_m128 b)

### SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally  
#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

## MOVLPD—Move Low Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD xmm1, m64	A	V/V	SSE2	Move double-precision floating-point value from m64 to low quadword of xmm1.
VEX.NDS.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	B	V/V	AVX	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
EVEX.NDS.128.66.0F.W1 12 /r VMOVLPD xmm2, xmm1, m64	D	V/V	AVX512F	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
66 0F 13/r MOVLPD m64, xmm1	C	V/V	SSE2	Move double-precision floating-point value from low quadword of xmm1 to m64.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	C	V/V	AVX	Move double-precision floating-point value from low quadword of xmm1 to m64.
EVEX.128.66.0F.W1 13/r VMOVLPD m64, xmm1	E	V/V	AVX512F	Move double-precision floating-point value from low quadword of xmm1 to m64.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

### Operation

#### MOVLPD (128-bit Legacy SSE load)

DEST[63:0] ← SRC[63:0]

DEST[MAXVL-1:64] (Unmodified)

**VMOVLPD (VEX.128 & EVEX encoded load)**

$$\text{DEST}[63:0] \leftarrow \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$$

$$\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$$
**VMOVLPD (store)**

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$
**Intel C/C++ Compiler Intrinsic Equivalent**
`MOVLPD __m128d _mm_loadl_pd ( __m128d a, double *p)`
`MOVLPD void _mm_storel_pd (double *p, __m128d a)`
**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

## MOVLPS—Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVLPS xmm1, m64	A	V/V	SSE	Move two packed single-precision floating-point values from m64 to low quadword of xmm1.
VEX.NDS.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
EVEX.NDS.128.0F.W0 12 /r VMOVLPS xmm2, xmm1, m64	D	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	C	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	C	V/V	AVX	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
EVEX.128.0F.W0 13/r VMOVLPS m64, xmm1	E	V/V	AVX512F	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

This instruction cannot be used for register to register or memory to memory moves.

#### 128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

#### VEX.128 & EVEX encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

#### 128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

**Operation****MOVLPS (128-bit Legacy SSE load)**

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$

$$\text{DEST}[\text{MAXVL}-1:64] \text{ (Unmodified)}$$
**VMOVLPS (VEX.128 & EVEX encoded load)**

$$\text{DEST}[63:0] \leftarrow \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64]$$

$$\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$$
**VMOVLPS (store)**

$$\text{DEST}[63:0] \leftarrow \text{SRC}[63:0]$$
**Intel C/C++ Compiler Intrinsic Equivalent**

$$\text{MOVLPS } \_m128 \_mm\_load\_pi ( \_m128 \text{ a}, \_m64 *p)$$

$$\text{MOVLPS void } \_mm\_store\_pi ( \_m64 *p, \_m128 \text{ a})$$
**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.



**MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 50 /r MOVMSKPD <i>reg, xmm</i>	RM	V/V	SSE2	Extract 2-bit sign mask from <i>xmm</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are filled with zeros.
VEX.128.66.0F.WIG 50 /r VMOVMSKPD <i>reg, xmm2</i>	RM	V/V	AVX	Extract 2-bit sign mask from <i>xmm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
VEX.256.66.0F.WIG 50 /r VMOVMSKPD <i>reg, ymm2</i>	RM	V/V	AVX	Extract 4-bit sign mask from <i>ymm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Extracts the sign bits from the packed double-precision floating-point values in the source operand (second operand), formats them into a 2-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand. Zero-extend the upper bits of the destination.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation****(V)MOVMSKPD (128-bit versions)**

DEST[0] ← SRC[63]

DEST[1] ← SRC[127]

IF DEST = r32

THEN DEST[31:2] ← 0;

ELSE DEST[63:2] ← 0;

FI

**VMOVMSKPD (VEX.256 encoded version)**

DEST[0] ← SRC[63]

DEST[1] ← SRC[127]

DEST[2] ← SRC[191]

DEST[3] ← SRC[255]

IF DEST = r32

THEN DEST[31:4] ← 0;

ELSE DEST[63:4] ← 0;

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

MOVMSKPD:       int \_mm\_movemask\_pd ( \_\_m128d a)

VMOVMSKPD:       \_mm256\_movemask\_pd(\_\_m256d a)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 7; additionally

#UD               If VEX.vvvv ≠ 1111B.

## MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP OF 50 /r MOVMSKPS <i>reg, xmm</i>	RM	V/V	SSE	Extract 4-bit sign mask from <i>xmm</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are filled with zeros.
VEX.128.OF.WIG 50 /r VMOVMSKPS <i>reg, xmm2</i>	RM	V/V	AVX	Extract 4-bit sign mask from <i>xmm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.
VEX.256.OF.WIG 50 /r VMOVMSKPS <i>reg, ymm2</i>	RM	V/V	AVX	Extract 8-bit sign mask from <i>ymm2</i> and store in <i>reg</i> . The upper bits of <i>r32</i> or <i>r64</i> are zeroed.

### Instruction Operand Encoding<sup>1</sup>

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

DEST[0] ← SRC[31];

DEST[1] ← SRC[63];

DEST[2] ← SRC[95];

DEST[3] ← SRC[127];

IF DEST = r32

THEN DEST[31:4] ← ZeroExtend;

ELSE DEST[63:4] ← ZeroExtend;

FI;

1. ModRM.MOD = 011B required

**(V)MOVMSKPS (128-bit version)**

```

DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI

```

**VMOVMSKPS (VEX.256 encoded version)**

```

DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
DEST[4] ← SRC[159]
DEST[5] ← SRC[191]
DEST[6] ← SRC[223]
DEST[7] ← SRC[255]
IF DEST = r32
    THEN DEST[31:8] ← 0;
    ELSE DEST[63:8] ← 0;
FI

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

int _mm_movemask_ps(__m128 a)
int _mm256_movemask_ps(__m256 a)

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 7; additionally  
 #UD If VEX.vvvv ≠ 1111B.

**MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA xmm1, m128	A	V/V	SSE4_1	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA xmm1, m128	A	V/V	AVX	Move double quadword from m128 to xmm using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA ymm1, m256	A	V/V	AVX2	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.128.66.0F38.W0 2A /r VMOVNTDQA xmm1, m128	B	V/V	AVX512VL AVX512F	Move 128-bit data from m128 to xmm using non-temporal hint if WC memory type.
EVEX.256.66.0F38.W0 2A /r VMOVNTDQA ymm1, m256	B	V/V	AVX512VL AVX512F	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.512.66.0F38.W0 2A /r VMOVNTDQA zmm1, m512	B	V/V	AVX512F	Move 512-bit data from m512 to zmm using non-temporal hint if WC memory type.

**Instruction Operand Encoding<sup>1</sup>**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the nontemporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor

does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer’s Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor’s implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementa-

1. ModRM.MOD = 011B required

tion may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

### Operation

#### MOVNTDQA (128bit- Legacy SSE form)

DEST  $\leftarrow$  SRC

DEST[MAXVL-1:128] (Unmodified)

#### VMOVNTDQA (VEX.128 and EVEX.128 encoded form)

DEST  $\leftarrow$  SRC

DEST[MAXVL-1:128]  $\leftarrow$  0

#### VMOVNTDQA (VEX.256 and EVEX.256 encoded forms)

DEST[255:0]  $\leftarrow$  SRC[255:0]

DEST[MAXVL-1:256]  $\leftarrow$  0

#### VMOVNTDQA (EVEX.512 encoded form)

DEST[511:0]  $\leftarrow$  SRC[511:0]

DEST[MAXVL-1:512]  $\leftarrow$  0

### Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQA \_\_m512i \_\_mm512\_stream\_load\_si512(void \* p);

MOVNTDQA \_\_m128i \_\_mm\_stream\_load\_si128 (\_\_m128i \*p);

VMOVNTDQA \_\_m256i \_\_mm\_stream\_load\_si256 (\_\_m256i \*p);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1;

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

**MOVNTDQ—Store Packed Integers Using Non-Temporal Hint**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ m128, xmm1	A	V/V	SSE2	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ m128, xmm1	A	V/V	AVX	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ m256, ymm1	A	V/V	AVX	Move packed integer values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W0 E7 /r VMOVNTDQ m128, xmm1	B	V/V	AVX512VL AVX512F	Move packed integer values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W0 E7 /r VMOVNTDQ m256, ymm1	B	V/V	AVX512VL AVX512F	Move packed integer values in zmm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W0 E7 /r VMOVNTDQ m512, zmm1	B	V/V	AVX512F	Move packed integer values in zmm1 to m512 using non-temporal hint.

**Instruction Operand Encoding<sup>1</sup>**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

**Description**

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, double-words, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

**Operation****VMOVNTDQ(EVEX encoded versions)**

VL = 128, 256, 512

DEST[VL-1:0] ← SRC[VL-1:0]

DEST[MAXVL-1:VL] ← 0

1. ModRM.MOD = 011B required

**MOVNTDQ (Legacy and VEX versions)**

DEST ← SRC

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVNTDQ void \_mm512\_stream\_si512(void \* p, \_\_m512i a);

VMOVNTDQ void \_mm256\_stream\_si256 (\_\_m256i \* p, \_\_m256i a);

MOVNTDQ void \_mm\_stream\_si128 (\_\_m128i \* p, \_\_m128i a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.



## MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF C3 /r	MOVNTI <i>m32, r32</i>	MR	Valid	Valid	Move doubleword from <i>r32</i> to <i>m32</i> using non-temporal hint.
NP REX.W + OF C3 /r	MOVNTI <i>m64, r64</i>	MR	Valid	N.E.	Move quadword from <i>r64</i> to <i>m64</i> using non-temporal hint.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, the instruction’s default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← SRC;

### Intel C/C++ Compiler Intrinsic Equivalent

MOVNTI:     void \_mm\_stream\_si32 (int \*p, int a)

MOVNTI:     void \_mm\_stream\_si64(\_\_int64 \*p, \_\_int64 a)

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

- #GP(0)           For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
- #SS(0)           For an illegal address in the SS segment.
- #PF(fault-code)   For a page fault.
- #UD              If CPUID.01H:EDX.SSE2[bit 26] = 0.
- If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0. If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
-----------------	-------------------

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0. If the LOCK prefix is used.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD m128, xmm1	A	V/V	SSE2	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD m128, xmm1	A	V/V	AVX	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD m256, ymm1	A	V/V	AVX	Move packed double-precision values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W1 2B /r VMOVNTPD m128, xmm1	B	V/V	AVX512VL AVX512F	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W1 2B /r VMOVNTPD m256, ymm1	B	V/V	AVX512VL AVX512F	Move packed double-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W1 2B /r VMOVNTPD m512, zmm1	B	V/V	AVX512F	Move packed double-precision values in zmm1 to m512 using non-temporal hint.

**Instruction Operand Encoding<sup>1</sup>**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

**Description**

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

**Operation****VMOVNTPD (EVEX encoded versions)**

VL = 128, 256, 512

DEST[VL-1:0] ← SRC[VL-1:0]

DEST[MAXVL-1:VL] ← 0

1. ModRM.MOD = 011B required

**MOVNTPD (Legacy and VEX versions)**

DEST ← SRC

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVNTPD void \_mm512\_stream\_pd(double \* p, \_\_m512d a);

VMOVNTPD void \_mm256\_stream\_pd (double \* p, \_\_m256d a);

MOVNTPD void \_mm\_stream\_pd (double \* p, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

**MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2B /r MOVNTPS m128, xmm1	A	V/V	SSE	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.128.0F.WIG 2B /r VMOVNTPS m128, xmm1	A	V/V	AVX	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.256.0F.WIG 2B /r VMOVNTPS m256, ymm1	A	V/V	AVX	Move packed single-precision values ymm1 to mem using non-temporal hint.
EVEX.128.0F.W0 2B /r VMOVNTPS m128, xmm1	B	V/V	AVX512VL AVX512F	Move packed single-precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.0F.W0 2B /r VMOVNTPS m256, ymm1	B	V/V	AVX512VL AVX512F	Move packed single-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.0F.W0 2B /r VMOVNTPS m512, zmm1	B	V/V	AVX512F	Move packed single-precision values in zmm1 to m512 using non-temporal hint.

**Instruction Operand Encoding<sup>1</sup>**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

**Description**

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the IA-32 Intel Architecture Software Developer’s Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

**Operation****VMOVNTPS (EVEX encoded versions)**

VL = 128, 256, 512

DEST[VL-1:0] ← SRC[VL-1:0]

DEST[MAXVL-1:VL] ← 0

1. ModRM.MOD = 011B required

**MOVNTPS**

DEST ← SRC

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVNTPS void \_mm512\_stream\_ps(float \* p, \_\_m512d a);

MOVNTPS void \_mm\_stream\_ps (float \* p, \_\_m128d a);

VMOVNTPS void \_mm256\_stream\_ps (float \* p, \_\_m256 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## MOVNTQ—Store of Quadword Using Non-Temporal Hint

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF E7 /r	MOVNTQ <i>m64, mm</i>	MR	Valid	Valid	Move quadword from <i>mm</i> to <i>m64</i> using non-temporal hint.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see “Caching of Temporal vs. Non-Temporal Data” in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

DEST ← SRC;

### Intel C/C++ Compiler Intrinsic Equivalent

MOVNTQ:     void \_mm\_stream\_pi(\_\_m64 \*p, \_\_m64 a)

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6F /r MOVQ <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Move quadword from <i>mm/m64</i> to <i>mm</i> .
NP 0F 7F /r MOVQ <i>mm/m64</i> , <i>mm</i>	B	V/V	MMX	Move quadword from <i>mm</i> to <i>mm/m64</i> .
F3 0F 7E /r MOVQ <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	SSE2	Move quadword from <i>xmm2/mem64</i> to <i>xmm1</i> .
VEX.128.F3.0F.WIG 7E /r VMOVQ <i>xmm1</i> , <i>xmm2/m64</i>	A	V/V	AVX	Move quadword from <i>xmm2</i> to <i>xmm1</i> .
EVEX.128.F3.0F.W1 7E /r VMOVQ <i>xmm1</i> , <i>xmm2/m64</i>	C	V/V	AVX512F	Move quadword from <i>xmm2/m64</i> to <i>xmm1</i> .
66 0F D6 /r MOVQ <i>xmm2/m64</i> , <i>xmm1</i>	B	V/V	SSE2	Move quadword from <i>xmm1</i> to <i>xmm2/mem64</i> .
VEX.128.66.0F.WIG D6 /r VMOVQ <i>xmm1/m64</i> , <i>xmm2</i>	B	V/V	AVX	Move quadword from <i>xmm2</i> register to <i>xmm1/m64</i> .
EVEX.128.66.0F.W1 D6 /r VMOVQ <i>xmm1/m64</i> , <i>xmm2</i>	D	V/V	AVX512F	Move quadword from <i>xmm2</i> register to <i>xmm1/m64</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.



## Operation

**MOVQ instruction when operating on MMX technology registers and memory locations**

$DEST \leftarrow SRC;$

**MOVQ instruction when source and destination operands are XMM registers**

$DEST[63:0] \leftarrow SRC[63:0];$

$DEST[127:64] \leftarrow 0000000000000000H;$

**MOVQ instruction when source operand is XMM register and destination**

operand is memory location:

$DEST \leftarrow SRC[63:0];$

**MOVQ instruction when source operand is memory location and destination**

operand is XMM register:

$DEST[63:0] \leftarrow SRC;$

$DEST[127:64] \leftarrow 0000000000000000H;$

**VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination**

$DEST[63:0] \leftarrow SRC[63:0]$

$DEST[MAXVL-1:64] \leftarrow 0$

**VMOVQ (VEX.128.66.0F D6) with XMM register source and destination**

$DEST[63:0] \leftarrow SRC[63:0]$

$DEST[MAXVL-1:64] \leftarrow 0$

**VMOVQ (7E - EVEX encoded version) with XMM register source and destination**

$DEST[63:0] \leftarrow SRC[63:0]$

$DEST[MAXVL-1:64] \leftarrow 0$

**VMOVQ (D6 - EVEX encoded version) with XMM register source and destination**

$DEST[63:0] \leftarrow SRC[63:0]$

$DEST[MAXVL-1:64] \leftarrow 0$

**VMOVQ (7E) with memory source**

$DEST[63:0] \leftarrow SRC[63:0]$

$DEST[MAXVL-1:64] \leftarrow 0$

**VMOVQ (7E - EVEX encoded version) with memory source**

$DEST[63:0] \leftarrow SRC[63:0]$

$DEST[MAXVL-1:64] \leftarrow 0$

**VMOVQ (D6) with memory dest**

$DEST[63:0] \leftarrow SRC2[63:0]$

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

VMOVQ \_\_m128i \_mm\_loadu\_si64( void \* s);

VMOVQ void \_mm\_storeu\_si64( void \* d, \_\_m128i s);

MOVQ            m128i \_mm\_mov\_epi64(\_\_m128i a)

## SIMD Floating-Point Exceptions

None

## Other Exceptions

See Table 22-8, “Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*.

## MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F3 0F D6 /r	MOVQ2DQ <i>xmm, mm</i>	RM	Valid	Valid	Move quadword from <i>mmx</i> to low quadword of <i>xmm</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ2DQ instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

### Operation

DEST[63:0] ← SRC[63:0];

DEST[127:64] ← 0000000000000000H;

### Intel C/C++ Compiler Intrinsic Equivalent

MOVQ2DQ: `__128i_mm_movpi64_pi64 ( __m64 a)`

### SIMD Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM If CR0.TS[bit 3] = 1.  
 #UD If CR0.EM[bit 2] = 1.  
 If CR4.OSFXSR[bit 9] = 0.  
 If CPUID.01H:EDX.SSE2[bit 26] = 0.  
 If the LOCK prefix is used.  
 #MF If there is a pending x87 FPU exception.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## MOVS/MOVS<sub>B</sub>/MOVSW/MOVS<sub>D</sub>/MOVSQ—Move Data from String to String

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
A4	MOVS <i>m8, m8</i>	Z0	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R)ESI to (R)EDI.
A5	MOVS <i>m16, m16</i>	Z0	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R)ESI to (R)EDI.
A5	MOVS <i>m32, m32</i>	Z0	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R)ESI to (R)EDI.
REX.W + A5	MOVS <i>m64, m64</i>	Z0	Valid	N.E.	Move qword from address (R)ESI to (R)EDI.
A4	MOVS <sub>B</sub>	Z0	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R)ESI to (R)EDI.
A5	MOVSW	Z0	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R)ESI to (R)EDI.
A5	MOVS <sub>D</sub>	Z0	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R)ESI to (R)EDI.
REX.W + A5	MOVSQ	Z0	Valid	N.E.	Move qword from address (R)ESI to (R)EDI.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). Both the source and destination operands are located in memory. The address of the source operand is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the destination operand is read from the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the MOVS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source and destination operands should be symbols that indicate the size and location of the source value and the destination, respectively. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source and destination operand symbols must specify the correct **type** (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct **location**. The locations of the source and destination operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the move string instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the MOVS instructions. Here also DS:(E)SI and ES:(E)DI are assumed to be the source and destination operands, respectively. The size of the source and destination operands is selected with the mnemonic: MOVS<sub>B</sub> (byte move), MOVSW (word move), or MOVS<sub>D</sub> (doubleword move).

After the move operation, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incre-

mented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

### NOTE

To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with MOVS and MOVSB. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

The MOVS, MOVSB, MOVSW, and MOVSD instructions can be preceded by the REP prefix (see "REP/REPE/REPZ/REPNE/REPZ—Repeat String Operation Prefix" for a description of the REP prefix) for block moves of ECX bytes, words, or doublewords.

In 64-bit mode, the instruction's default address size is 64 bits, 32-bit address size is supported using the prefix 67H. The 64-bit addresses are specified by RSI and RDI; 32-bit address are specified by ESI and EDI. Use of the REX.W prefix promotes doubleword operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST  $\leftarrow$  SRC;

Non-64-bit Mode:

```
IF (Byte move)
  THEN IF DF = 0
    THEN
      (E)SI  $\leftarrow$  (E)SI + 1;
      (E)DI  $\leftarrow$  (E)DI + 1;
    ELSE
      (E)SI  $\leftarrow$  (E)SI - 1;
      (E)DI  $\leftarrow$  (E)DI - 1;
    FI;
  ELSE IF (Word move)
    THEN IF DF = 0
      (E)SI  $\leftarrow$  (E)SI + 2;
      (E)DI  $\leftarrow$  (E)DI + 2;
      FI;
    ELSE
      (E)SI  $\leftarrow$  (E)SI - 2;
      (E)DI  $\leftarrow$  (E)DI - 2;
      FI;
  ELSE IF (Doubleword move)
    THEN IF DF = 0
      (E)SI  $\leftarrow$  (E)SI + 4;
      (E)DI  $\leftarrow$  (E)DI + 4;
      FI;
    ELSE
      (E)SI  $\leftarrow$  (E)SI - 4;
      (E)DI  $\leftarrow$  (E)DI - 4;
      FI;
  FI;
```

64-bit Mode:

```
IF (Byte move)
  THEN IF DF = 0
    THEN
```

```

        (R|E)SI ← (R|E)SI + 1;
        (R|E)DI ← (R|E)DI + 1;
    ELSE
        (R|E)SI ← (R|E)SI - 1;
        (R|E)DI ← (R|E)DI - 1;
    FI;
ELSE IF (Word move)
    THEN IF DF = 0
        (R|E)SI ← (R|E)SI + 2;
        (R|E)DI ← (R|E)DI + 2;
        FI;
    ELSE
        (R|E)SI ← (R|E)SI - 2;
        (R|E)DI ← (R|E)DI - 2;
    FI;
ELSE IF (Doubleword move)
    THEN IF DF = 0
        (R|E)SI ← (R|E)SI + 4;
        (R|E)DI ← (R|E)DI + 4;
        FI;
    ELSE
        (R|E)SI ← (R|E)SI - 4;
        (R|E)DI ← (R|E)DI - 4;
    FI;
ELSE IF (Quadword move)
    THEN IF DF = 0
        (R|E)SI ← (R|E)SI + 8;
        (R|E)DI ← (R|E)DI + 8;
        FI;
    ELSE
        (R|E)SI ← (R|E)SI - 8;
        (R|E)DI ← (R|E)DI - 8;
    FI;
FI;

```

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register contains a NULL segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## MOVSD—Move or Merge Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 10 /r MOVSD xmm1, xmm2	A	V/V	SSE2	Move scalar double-precision floating-point value from xmm2 to xmm1 register.
F2 0F 10 /r MOVSD xmm1, m64	A	V/V	SSE2	Load scalar double-precision floating-point value from m64 to xmm1 register.
F2 0F 11 /r MOVSD xmm1/m64, xmm2	C	V/V	SSE2	Move scalar double-precision floating-point value from xmm2 register to xmm1/m64.
VEX.NDS.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, m64	D	V/V	AVX	Load scalar double-precision floating-point value from m64 to xmm1 register.
VEX.NDS.LIG.F2.0F.WIG 11 /r VMOVSD xmm1, xmm2, xmm3	E	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD m64, xmm1	C	V/V	AVX	Store scalar double-precision floating-point value from xmm1 register to m64.
EVEX.NDS.LIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, m64	F	V/V	AVX512F	Load scalar double-precision floating-point value from m64 to xmm1 register under writemask k1.
EVEX.NDS.LIG.F2.0F.W1 11 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.0F.W1 11 /r VMOVSD m64 {k1}, xmm1	G	V/V	AVX512F	Store scalar double-precision floating-point value from xmm1 register to m64 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	NA	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA



**Description**

Moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits MAXVL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar double-precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:64 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low quadword of the destination is updated according to the writemask.

Note: For VMOVS D (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

**Operation****VMOVS D (EVEX.NDS.LIG.F2.0F 10 /r: VMOVS D xmm1, m64 with support for 32 registers)**

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← SRC[63:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[MAXVL-1:64] ← 0

**VMOVS D (EVEX.NDS.LIG.F2.0F 11 /r: VMOVS D m64, xmm1 with support for 32 registers)**

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← SRC[63:0]

ELSE \*DEST[63:0] remains unchanged\* ; merging-masking

FI;

**VMOVS D (EVEX.NDS.LIG.F2.0F 11 /r: VMOVS D xmm1, xmm2, xmm3)**

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← SRC2[63:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAXVL-1:128] ← 0

**MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)**

DEST[63:0] ← SRC[63:0]  
 DEST[MAXVL-1:64] (Unmodified)

**VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)**

DEST[63:0] ← SRC2[63:0]  
 DEST[127:64] ← SRC1[127:64]  
 DEST[MAXVL-1:128] ← 0

**VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)**

DEST[63:0] ← SRC2[63:0]  
 DEST[127:64] ← SRC1[127:64]  
 DEST[MAXVL-1:128] ← 0

**VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)**

DEST[63:0] ← SRC[63:0]  
 DEST[MAXVL-1:64] ← 0

**MOVSD/VMOVSD (128-bit versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)**

DEST[63:0] ← SRC[63:0]

**MOVSD (128-bit Legacy SSE version: MOVSD XMM1, m64)**

DEST[63:0] ← SRC[63:0]  
 DEST[127:64] ← 0  
 DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSD \_\_m128d \_mm\_mask\_load\_sd(\_\_m128d s, \_\_mmask8 k, double \* p);  
 VMOVSD \_\_m128d \_mm\_maskz\_load\_sd(\_\_mmask8 k, double \* p);  
 VMOVSD \_\_m128d \_mm\_mask\_move\_sd(\_\_m128d sh, \_\_mmask8 k, \_\_m128d sl, \_\_m128d a);  
 VMOVSD \_\_m128d \_mm\_maskz\_move\_sd(\_\_mmask8 k, \_\_m128d s, \_\_m128d a);  
 VMOVSD void \_mm\_mask\_store\_sd(double \* p, \_\_mmask8 k, \_\_m128d s);  
 MOVSD \_\_m128d \_mm\_load\_sd(double \* p)  
 MOVSD void \_mm\_store\_sd(double \* p, \_\_m128d a)  
 MOVSD \_\_m128d \_mm\_move\_sd(\_\_m128d a, \_\_m128d b)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

## MOVSHDUP—Replicate Single FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128	A	V/V	SSE3	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP xmm1, xmm2/m128	A	V/V	AVX	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP ymm1, ymm2/m256	A	V/V	AVX	Move odd index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 16 /r VMOVSHDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Move odd index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 16 /r VMOVSHDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512F	Move odd index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 16 /r VMOVSHDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F	Move odd index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Duplicates odd-indexed single-precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 4-3. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

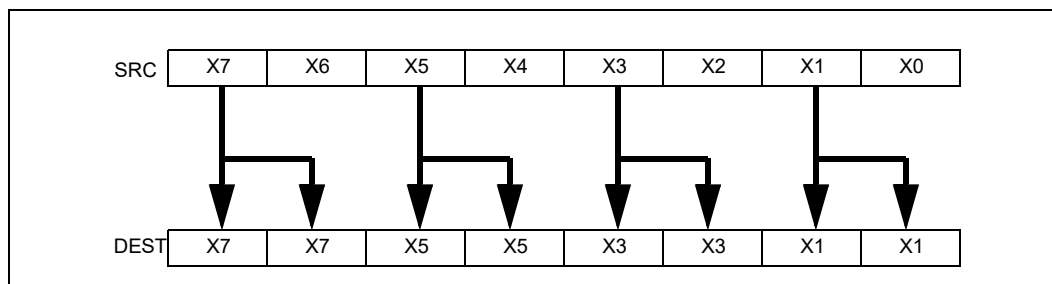


Figure 4-3. MOVSHDUP Operation

**Operation****VMOVSHDUP (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP\_SRC[31:0] ← SRC[63:32]

TMP\_SRC[63:32] ← SRC[63:32]

TMP\_SRC[95:64] ← SRC[127:96]

TMP\_SRC[127:96] ← SRC[127:96]

IF VL ≥ 256

    TMP\_SRC[159:128] ← SRC[191:160]

    TMP\_SRC[191:160] ← SRC[191:160]

    TMP\_SRC[223:192] ← SRC[255:224]

    TMP\_SRC[255:224] ← SRC[255:224]

FI;

IF VL ≥ 512

    TMP\_SRC[287:256] ← SRC[319:288]

    TMP\_SRC[319:288] ← SRC[319:288]

    TMP\_SRC[351:320] ← SRC[383:352]

    TMP\_SRC[383:352] ← SRC[383:352]

    TMP\_SRC[415:384] ← SRC[447:416]

    TMP\_SRC[447:416] ← SRC[447:416]

    TMP\_SRC[479:448] ← SRC[511:480]

    TMP\_SRC[511:480] ← SRC[511:480]

FI;

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+31:i] ← TMP\_SRC[i+31:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

            DEST[i+31:i] ← 0

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVSHDUP (VEX.256 encoded version)**

DEST[31:0] ← SRC[63:32]

DEST[63:32] ← SRC[63:32]

DEST[95:64] ← SRC[127:96]

DEST[127:96] ← SRC[127:96]

DEST[159:128] ← SRC[191:160]

DEST[191:160] ← SRC[191:160]

DEST[223:192] ← SRC[255:224]

DEST[255:224] ← SRC[255:224]

DEST[MAXVL-1:256] ← 0

**VMOVSHDUP (VEX.128 encoded version)**

DEST[31:0] ← SRC[63:32]

DEST[63:32] ← SRC[63:32]

DEST[95:64] ← SRC[127:96]

DEST[127:96] ← SRC[127:96]

DEST[MAXVL-1:128] ← 0

**MOVSHDUP (128-bit Legacy SSE version)**

DEST[31:0] ← SRC[63:32]

DEST[63:32] ← SRC[63:32]

DEST[95:64] ← SRC[127:96]

DEST[127:96] ← SRC[127:96]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSHDUP \_\_m512 \_\_mm512\_movehdup\_ps( \_\_m512 a);

VMOVSHDUP \_\_m512 \_\_mm512\_mask\_movehdup\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VMOVSHDUP \_\_m512 \_\_mm512\_maskz\_movehdup\_ps( \_\_mmask16 k, \_\_m512 a);

VMOVSHDUP \_\_m256 \_\_mm256\_mask\_movehdup\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VMOVSHDUP \_\_m256 \_\_mm256\_maskz\_movehdup\_ps( \_\_mmask8 k, \_\_m256 a);

VMOVSHDUP \_\_m128 \_\_mm\_mask\_movehdup\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VMOVSHDUP \_\_m128 \_\_mm\_maskz\_movehdup\_ps( \_\_mmask8 k, \_\_m128 a);

VMOVSHDUP \_\_m256 \_\_mm256\_movehdup\_ps (\_\_m256 a);

VMOVSHDUP \_\_m128 \_\_mm\_movehdup\_ps (\_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

## MOVSLDUP—Replicate Single FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128	A	V/V	SSE3	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP xmm1, xmm2/m128	A	V/V	AVX	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 12 /r VMOVSLDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Move even index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 12 /r VMOVSLDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512F	Move even index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 12 /r VMOVSLDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F	Move even index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Duplicates even-indexed single-precision floating-point values from the source operand (the second operand). See Figure 4-4. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

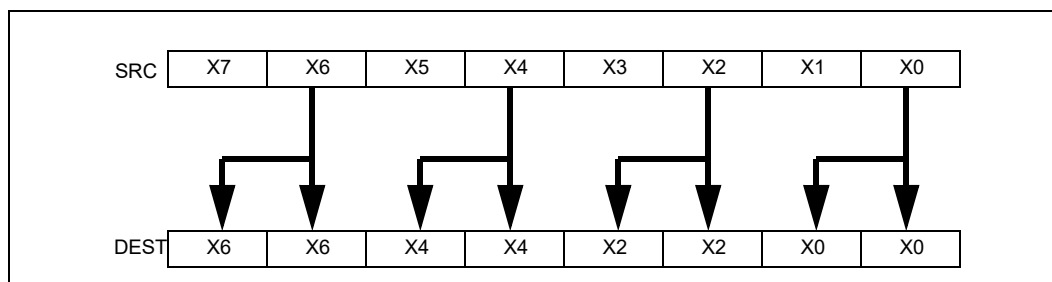


Figure 4-4. MOVSLDUP Operation

**Operation****VMOVSLDUP (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

TMP\_SRC[31:0] ← SRC[31:0]

TMP\_SRC[63:32] ← SRC[31:0]

TMP\_SRC[95:64] ← SRC[95:64]

TMP\_SRC[127:96] ← SRC[95:64]

IF VL ≥ 256

TMP\_SRC[159:128] ← SRC[159:128]

TMP\_SRC[191:160] ← SRC[159:128]

TMP\_SRC[223:192] ← SRC[223:192]

TMP\_SRC[255:224] ← SRC[223:192]

FI;

IF VL ≥ 512

TMP\_SRC[287:256] ← SRC[287:256]

TMP\_SRC[319:288] ← SRC[287:256]

TMP\_SRC[351:320] ← SRC[351:320]

TMP\_SRC[383:352] ← SRC[351:320]

TMP\_SRC[415:384] ← SRC[415:384]

TMP\_SRC[447:416] ← SRC[415:384]

TMP\_SRC[479:448] ← SRC[479:448]

TMP\_SRC[511:480] ← SRC[479:448]

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVSLDUP (VEX.256 encoded version)**

DEST[31:0] ← SRC[31:0]

DEST[63:32] ← SRC[31:0]

DEST[95:64] ← SRC[95:64]

DEST[127:96] ← SRC[95:64]

DEST[159:128] ← SRC[159:128]

DEST[191:160] ← SRC[159:128]

DEST[223:192] ← SRC[223:192]

DEST[255:224] ← SRC[223:192]

DEST[MAXVL-1:256] ← 0

**VMOVSLDUP (VEX.128 encoded version)**

DEST[31:0] ← SRC[31:0]

DEST[63:32] ← SRC[31:0]

DEST[95:64] ← SRC[95:64]

DEST[127:96] ← SRC[95:64]

DEST[MAXVL-1:128] ← 0

**MOVSLDUP (128-bit Legacy SSE version)**

DEST[31:0] ← SRC[31:0]  
 DEST[63:32] ← SRC[31:0]  
 DEST[95:64] ← SRC[95:64]  
 DEST[127:96] ← SRC[95:64]  
 DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVSLDUP \_\_m512 \_\_mm512\_moveldup\_ps( \_\_m512 a);  
 VMOVSLDUP \_\_m512 \_\_mm512\_mask\_moveldup\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512 a);  
 VMOVSLDUP \_\_m512 \_\_mm512\_maskz\_moveldup\_ps( \_\_mmask16 k, \_\_m512 a);  
 VMOVSLDUP \_\_m256 \_\_mm256\_mask\_moveldup\_ps( \_\_m256 s, \_\_mmask8 k, \_\_m256 a);  
 VMOVSLDUP \_\_m256 \_\_mm256\_maskz\_moveldup\_ps( \_\_mmask8 k, \_\_m256 a);  
 VMOVSLDUP \_\_m128 \_\_mm\_mask\_moveldup\_ps( \_\_m128 s, \_\_mmask8 k, \_\_m128 a);  
 VMOVSLDUP \_\_m128 \_\_mm\_maskz\_moveldup\_ps( \_\_mmask8 k, \_\_m128 a);  
 VMOVSLDUP \_\_m256 \_\_mm256\_moveldup\_ps( \_\_m256 a);  
 VMOVSLDUP \_\_m128 \_\_mm\_moveldup\_ps( \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

#UD If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.



## MOVSS—Move or Merge Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS xmm1, xmm2	A	V/V	SSE	Merge scalar single-precision floating-point value from xmm2 to xmm1 register.
F3 0F 10 /r MOVSS xmm1, m32	A	V/V	SSE	Load scalar single-precision floating-point value from m32 to xmm1 register.
VEX.NDS.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32	D	V/V	AVX	Load scalar single-precision floating-point value from m32 to xmm1 register.
F3 0F 11 /r MOVSS xmm2/m32, xmm1	C	V/V	SSE	Move scalar single-precision floating-point value from xmm1 register to xmm2/m32.
VEX.NDS.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3	E	V/V	AVX	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS m32, xmm1	C	V/V	AVX	Move scalar single-precision floating-point value from xmm1 register to m32.
EVEX.NDS.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, m32	F	V/V	AVX512F	Move scalar single-precision floating-point values from m32 to xmm1 under writemask k1.
EVEX.NDS.LIG.F3.0F.W0 11 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 11 /r VMOVSS m32 {k1}, xmm1	G	V/V	AVX512F	Move scalar single-precision floating-point values from xmm1 to m32 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	NA	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

## Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAXVL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination operand is an XMM registers, Bits (127:32) of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single-precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:32 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low doubleword of the destination is updated according to the writemask.

Note: For memory store form instruction "VMOVSS m32, xmm1", VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction "VMOVSS mv {k1}, xmm1", EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Software should ensure VMOVSS is encoded with VEX.L=0. Encoding VMOVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

## Operation

**VMOVSS (EVEX.NDS.LIG.F3.OF.W0 11 /r when the source operand is memory and the destination is an XMM register)**

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← SRC[31:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0] ← 0

FI;

FI;

DEST[MAXVL-1:32] ← 0

**VMOVSS (EVEX.NDS.LIG.F3.OF.W0 10 /r when the source operand is an XMM register and the destination is memory)**

IF k1[0] or \*no writemask\*

THEN DEST[31:0] ← SRC[31:0]

ELSE \*DEST[31:0] remains unchanged\* ; merging-masking

FI;

**VMOVSS (EVEX.NDS.LIG.F3.0F.W0 10/11 /r where the source and destination are XMM registers)**

```

IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                               ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)**

```

DEST[31:0] ← SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

```

**VMOVSS (VEX.NDS.128.F3.0F 11 /r where the destination is an XMM register)**

```

DEST[31:0] ← SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**VMOVSS (VEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)**

```

DEST[31:0] ← SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**VMOVSS (VEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)**

```

DEST[31:0] ← SRC[31:0]
DEST[MAXVL-1:32] ← 0

```

**MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)**

```

DEST[31:0] ← SRC[31:0]

```

**MOVSS (Legacy SSE version when the source operand is memory and the destination is an XMM register)**

```

DEST[31:0] ← SRC[31:0]
DEST[127:32] ← 0
DEST[MAXVL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMOVSS __m128 __mm_mask_load_ss(__m128 s, __mmask8 k, float * p);
VMOVSS __m128 __mm_maskz_load_ss(__mmask8 k, float * p);
VMOVSS __m128 __mm_mask_move_ss(__m128 sh, __mmask8 k, __m128 sl, __m128 a);
VMOVSS __m128 __mm_maskz_move_ss(__mmask8 k, __m128 s, __m128 a);
VMOVSS void __mm_mask_store_ss(float * p, __mmask8 k, __m128 a);
MOVSS __m128 __mm_load_ss(float * p)
MOVSS void __mm_store_ss(float * p, __m128 a)
MOVSS __m128 __mm_move_ss(__m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally  
#UD If VEX.vvvv != 1111B.  
EVEX-encoded instruction, see Exceptions Type E10.

## MOVSX/MOVSXD—Move with Sign-Extension

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF BE /r	MOVSX r16, r/m8	RM	Valid	Valid	Move byte to word with sign-extension.
OF BE /r	MOVSX r32, r/m8	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX.W + OF BE /r	MOVSX r64, r/m8	RM	Valid	N.E.	Move byte to quadword with sign-extension.
OF BF /r	MOVSX r32, r/m16	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + OF BF /r	MOVSX r64, r/m16	RM	Valid	N.E.	Move word to quadword with sign-extension.
63 /r*	MOVSXD r16, r/m16	RM	Valid	Valid	Move word to word with sign-extension.
63 /r*	MOVSXD r32, r/m32	RM	Valid	Valid	Move doubleword to doubleword with sign-extension.
REX.W + 63 /r	MOVSXD r64, r/m32	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

### NOTES:

\* The use of MOVSXD without REX.W in 64-bit mode is discouraged. Regular MOV should be used instead of using MOVSXD without REX.W.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← SignExtend(SRC);

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.
66 0F 11 /r MOVUPD xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 10 /r VMOVUPD xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 11 /r VMOVUPD xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 10 /r VMOVUPD ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed double-precision floating-point from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 11 /r VMOVUPD ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed double-precision floating-point from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 10 /r VMOVUPD xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from xmm2/m128 to xmm1 using writemask k1.
EVEX.128.66.0F.W1 11 /r VMOVUPD xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 10 /r VMOVUPD ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from ymm2/m256 to ymm1 using writemask k1.
EVEX.256.66.0F.W1 11 /r VMOVUPD ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 10 /r VMOVUPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.512.66.0F.W1 11 /r VMOVUPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### EVEX.512 encoded version:

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.

**VEX.256 encoded version:**

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

**128-bit versions:**

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

**VEX.128 and EVEX.128 encoded versions:** Bits (MAXVL-1:128) of the destination register are zeroed.

**Operation****VMOVUPD (EVEX encoded versions, register-copy form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← SRC[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE DEST[i+63:i] ← 0 ; zeroing-masking

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVUPD (EVEX encoded versions, store-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← SRC[i+63:i]

    ELSE \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR;



**VMOVUPD (EVEX encoded versions, load-form)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

VMOVUPD (VEX.256 encoded version, load - and register copy)

DEST[255:0] ← SRC[255:0]

DEST[MAXVL-1:256] ← 0

**VMOVUPD (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVUPD (VEX.128 encoded version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] ← 0

**MOVUPD (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)

**(V)MOVUPD (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVUPD \_\_m512d \_\_mm512\_loadu\_pd( void \* s);

VMOVUPD \_\_m512d \_\_mm512\_mask\_loadu\_pd(\_\_m512d a, \_\_mmask8 k, void \* s);

VMOVUPD \_\_m512d \_\_mm512\_maskz\_loadu\_pd( \_\_mmask8 k, void \* s);

VMOVUPD void \_\_mm512\_storeu\_pd( void \* d, \_\_m512d a);

VMOVUPD void \_\_mm512\_mask\_storeu\_pd( void \* d, \_\_mmask8 k, \_\_m512d a);

VMOVUPD \_\_m256d \_\_mm256\_mask\_loadu\_pd(\_\_m256d s, \_\_mmask8 k, void \* m);

VMOVUPD \_\_m256d \_\_mm256\_maskz\_loadu\_pd( \_\_mmask8 k, void \* m);

VMOVUPD void \_\_mm256\_mask\_storeu\_pd( void \* d, \_\_mmask8 k, \_\_m256d a);

VMOVUPD \_\_m128d \_\_mm\_mask\_loadu\_pd(\_\_m128d s, \_\_mmask8 k, void \* m);

VMOVUPD \_\_m128d \_\_mm\_maskz\_loadu\_pd( \_\_mmask8 k, void \* m);

VMOVUPD void \_\_mm\_mask\_storeu\_pd( void \* d, \_\_mmask8 k, \_\_m128d a);

MOVUPD \_\_m256d \_\_mm256\_loadu\_pd( double \* p);

MOVUPD void \_\_mm256\_storeu\_pd( double \*p, \_\_m256d a);

MOVUPD \_\_m128d \_\_mm\_loadu\_pd( double \* p);

MOVUPD void \_\_mm\_storeu\_pd( double \*p, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

Note treatment of #AC varies; additionally

#UD                      If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 10 /r MOVUPS xmm1, xmm2/m128	A	V/V	SSE	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
NP OF 11 /r MOVUPS xmm2/m128, xmm1	B	V/V	SSE	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.128.OF.WIG 10 /r VMOVUPS xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
VEX.128.OF 11.WIG /r VMOVUPS xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.256.OF 10.WIG /r VMOVUPS ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed single-precision floating-point from ymm2/mem to ymm1.
VEX.256.OF 11.WIG /r VMOVUPS ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed single-precision floating-point from ymm1 to ymm2/mem.
EVEX.128.OF.W0 10 /r VMOVUPS xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.OF.W0 10 /r VMOVUPS ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.OF.W0 10 /r VMOVUPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.OF.W0 11 /r VMOVUPS xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.OF.W0 11 /r VMOVUPS ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.OF.W0 11 /r VMOVUPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.

**VEX.256 and EVEX.256 encoded versions:**

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

**128-bit versions:**

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

**128-bit Legacy SSE version:** Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

**VEX.128 and EVEX.128 encoded versions:** Bits (MAXVL-1:128) of the destination register are zeroed.

**Operation****VMOVUPS (EVEX encoded versions, register-copy form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+31:i] ← SRC[i+31:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+31:i] remains unchanged\*

        ELSE DEST[i+31:i] ← 0 ; zeroing-masking

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVUPS (EVEX encoded versions, store-form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+31:i] ← SRC[i+31:i]

    ELSE \*DEST[i+31:i] remains unchanged\* ; merging-masking

    FI;

ENDFOR;

**VMOVUPS (EVEX encoded versions, load-form)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE DEST[i+31:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMOVUPS (VEX.256 encoded version, load - and register copy)**

DEST[255:0] ← SRC[255:0]

DEST[MAXVL-1:256] ← 0

**VMOVUPS (VEX.256 encoded version, store-form)**

DEST[255:0] ← SRC[255:0]

**VMOVUPS (VEX.128 encoded version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] ← 0

**MOVUPS (128-bit load- and register-copy- form Legacy SSE version)**

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)

**(V)MOVUPS (128-bit store-form version)**

DEST[127:0] ← SRC[127:0]

**Intel C/C++ Compiler Intrinsic Equivalent**

VMOVUPS \_\_m512 \_\_mm512\_loadu\_ps( void \* s);

VMOVUPS \_\_m512 \_\_mm512\_mask\_loadu\_ps(\_\_m512 a, \_\_mmask16 k, void \* s);

VMOVUPS \_\_m512 \_\_mm512\_maskz\_loadu\_ps( \_\_mmask16 k, void \* s);

VMOVUPS void \_\_mm512\_storeu\_ps( void \* d, \_\_m512 a);

VMOVUPS void \_\_mm512\_mask\_storeu\_ps( void \* d, \_\_mmask8 k, \_\_m512 a);

VMOVUPS \_\_m256 \_\_mm256\_mask\_loadu\_ps(\_\_m256 a, \_\_mmask8 k, void \* s);

VMOVUPS \_\_m256 \_\_mm256\_maskz\_loadu\_ps( \_\_mmask8 k, void \* s);

VMOVUPS void \_\_mm256\_mask\_storeu\_ps( void \* d, \_\_mmask8 k, \_\_m256 a);

VMOVUPS \_\_m128 \_\_mm\_mask\_loadu\_ps(\_\_m128 a, \_\_mmask8 k, void \* s);

VMOVUPS \_\_m128 \_\_mm\_maskz\_loadu\_ps( \_\_mmask8 k, void \* s);

VMOVUPS void \_\_mm\_mask\_storeu\_ps( void \* d, \_\_mmask8 k, \_\_m128 a);

MOVUPS \_\_m256 \_\_mm256\_loadu\_ps ( float \* p);

MOVUPS void \_\_mm256\_storeu\_ps( float \*p, \_\_m256 a);

MOVUPS \_\_m128 \_\_mm\_loadu\_ps ( float \* p);

MOVUPS void \_\_mm\_storeu\_ps( float \*p, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

Note treatment of #AC varies;

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD                      If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

## MOVZX—Move with Zero-Extend

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF B6 /r	MOVZX r16, r/m8	RM	Valid	Valid	Move byte to word with zero-extension.
OF B6 /r	MOVZX r32, r/m8	RM	Valid	Valid	Move byte to doubleword, zero-extension.
REX.W + OF B6 /r	MOVZX r64, r/m8*	RM	Valid	N.E.	Move byte to quadword, zero-extension.
OF B7 /r	MOVZX r32, r/m16	RM	Valid	Valid	Move word to doubleword, zero-extension.
REX.W + OF B7 /r	MOVZX r64, r/m16	RM	Valid	N.E.	Move word to quadword, zero-extension.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if the REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← ZeroExtend(SRC);

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.



## MPSADBW — Compute Multiple Packed Sums of Absolute Difference

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 42 /r ib MPSADBW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and writes the results in <i>xmm1</i> . Starting offsets within <i>xmm1</i> and <i>xmm2/m128</i> are determined by <i>imm8</i> .
VEX.NDS.128.66.0F3A.WIG 42 /r ib VMPSADBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and writes the results in <i>xmm1</i> . Starting offsets within <i>xmm2</i> and <i>xmm3/m128</i> are determined by <i>imm8</i> .
VEX.NDS.256.66.0F3A.WIG 42 /r ib VMPSADBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX2	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in <i>xmm2</i> and <i>ymm3/m128</i> and writes the results in <i>ymm1</i> . Starting offsets within <i>ymm2</i> and <i>xmm3/m128</i> are determined by <i>imm8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

(V)MPSADBW calculates packed word results of sum-absolute-difference (SAD) of unsigned bytes from two blocks of 32-bit dword elements, using two select fields in the immediate byte to select the offsets of the two blocks within the first source operand and the second operand. Packed SAD word results are calculated within each 128-bit lane. Each SAD word result is calculated between a stationary block\_2 (whose offset within the second source operand is selected by a two bit select control, multiplied by 32 bits) and a sliding block\_1 at consecutive byte-granular position within the first source operand. The offset of the first 32-bit block of block\_1 is selectable using a one bit select control, multiplied by 32 bits.

128-bit Legacy SSE version: Imm8[1:0]\*32 specifies the bit offset of block\_2 within the second source operand. Imm[2]\*32 specifies the initial bit offset of the block\_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Bits 7:3 of the immediate byte are ignored.

VEX.128 encoded version: Imm8[1:0]\*32 specifies the bit offset of block\_2 within the second source operand. Imm[2]\*32 specifies the initial bit offset of the block\_1 within the first source operand. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (127:128) of the corresponding YMM register are zeroed. Bits 7:3 of the immediate byte are ignored.

VEX.256 encoded version: The sum-absolute-difference (SAD) operation is repeated 8 times for MPSADW between the same block\_2 (fixed offset within the second source operand) and a variable block\_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations between block\_2 and block\_1 is written to the respective word in the lower 128 bits of the destination operand.

Additionally, VMPSADBW performs another eight SAD operations on block\_4 of the second source operand and block\_3 of the first source operand. (Imm8[4:3]\*32 + 128) specifies the bit offset of block\_4 within the second source operand. (Imm[5]\*32+128) specifies the initial bit offset of the block\_3 within the first source operand. Each 16-bit result of eight SAD operations between block\_4 and block\_3 is written to the respective word in the upper 128 bits of the destination operand.

The first source operand is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits 7:6 of the immediate byte are ignored.

Note: If VMPSADBW is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause a #UD exception.

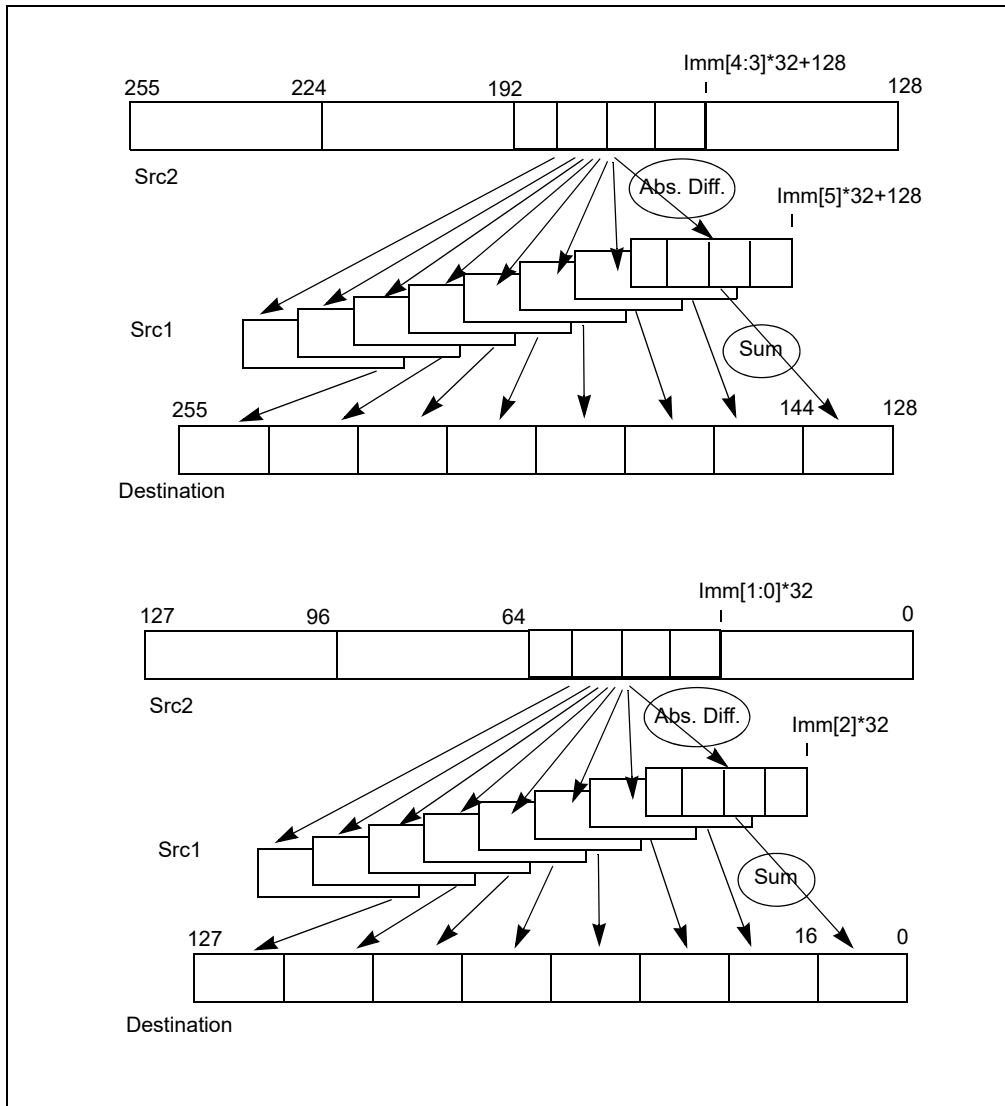


Figure 4-5. 256-bit VMPSADBW Operation

**Operation****VMPSADBW (VEX.256 encoded version)**

$$\text{BLK2\_OFFSET} \leftarrow \text{imm8}[1:0] * 32$$

$$\text{BLK1\_OFFSET} \leftarrow \text{imm8}[2] * 32$$

$$\text{SRC1\_BYTE0} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+7:\text{BLK1\_OFFSET}]$$

$$\text{SRC1\_BYTE1} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+15:\text{BLK1\_OFFSET}+8]$$

$$\text{SRC1\_BYTE2} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+23:\text{BLK1\_OFFSET}+16]$$

$$\text{SRC1\_BYTE3} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+31:\text{BLK1\_OFFSET}+24]$$

$$\text{SRC1\_BYTE4} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+39:\text{BLK1\_OFFSET}+32]$$

$$\text{SRC1\_BYTE5} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+47:\text{BLK1\_OFFSET}+40]$$

$$\text{SRC1\_BYTE6} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+55:\text{BLK1\_OFFSET}+48]$$

$$\text{SRC1\_BYTE7} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+63:\text{BLK1\_OFFSET}+56]$$

$$\text{SRC1\_BYTE8} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+71:\text{BLK1\_OFFSET}+64]$$

$$\text{SRC1\_BYTE9} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+79:\text{BLK1\_OFFSET}+72]$$

$$\text{SRC1\_BYTE10} \leftarrow \text{SRC1}[\text{BLK1\_OFFSET}+87:\text{BLK1\_OFFSET}+80]$$

$$\text{SRC2\_BYTE0} \leftarrow \text{SRC2}[\text{BLK2\_OFFSET}+7:\text{BLK2\_OFFSET}]$$

$$\text{SRC2\_BYTE1} \leftarrow \text{SRC2}[\text{BLK2\_OFFSET}+15:\text{BLK2\_OFFSET}+8]$$

$$\text{SRC2\_BYTE2} \leftarrow \text{SRC2}[\text{BLK2\_OFFSET}+23:\text{BLK2\_OFFSET}+16]$$

$$\text{SRC2\_BYTE3} \leftarrow \text{SRC2}[\text{BLK2\_OFFSET}+31:\text{BLK2\_OFFSET}+24]$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{SRC1\_BYTE0} - \text{SRC2\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{SRC1\_BYTE1} - \text{SRC2\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{SRC1\_BYTE2} - \text{SRC2\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{SRC1\_BYTE3} - \text{SRC2\_BYTE3})$$

$$\text{DEST}[15:0] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{SRC1\_BYTE1} - \text{SRC2\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{SRC1\_BYTE2} - \text{SRC2\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{SRC1\_BYTE3} - \text{SRC2\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{SRC1\_BYTE4} - \text{SRC2\_BYTE3})$$

$$\text{DEST}[31:16] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{SRC1\_BYTE2} - \text{SRC2\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{SRC1\_BYTE3} - \text{SRC2\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{SRC1\_BYTE4} - \text{SRC2\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{SRC1\_BYTE5} - \text{SRC2\_BYTE3})$$

$$\text{DEST}[47:32] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{SRC1\_BYTE3} - \text{SRC2\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{SRC1\_BYTE4} - \text{SRC2\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{SRC1\_BYTE5} - \text{SRC2\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{SRC1\_BYTE6} - \text{SRC2\_BYTE3})$$

$$\text{DEST}[63:48] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{SRC1\_BYTE4} - \text{SRC2\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{SRC1\_BYTE5} - \text{SRC2\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{SRC1\_BYTE6} - \text{SRC2\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{SRC1\_BYTE7} - \text{SRC2\_BYTE3})$$

$$\text{DEST}[79:64] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

$TEMP0 \leftarrow ABS(SRC1\_BYTE5 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE6 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE7 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE8 - SRC2\_BYTE3)$   
 $DEST[95:80] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE6 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE7 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE8 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE9 - SRC2\_BYTE3)$   
 $DEST[111:96] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE7 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE8 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE9 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE10 - SRC2\_BYTE3)$   
 $DEST[127:112] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$BLK2\_OFFSET \leftarrow imm8[4:3]*32 + 128$   
 $BLK1\_OFFSET \leftarrow imm8[5]*32 + 128$   
 $SRC1\_BYTE0 \leftarrow SRC1[BLK1\_OFFSET+7:BLK1\_OFFSET]$   
 $SRC1\_BYTE1 \leftarrow SRC1[BLK1\_OFFSET+15:BLK1\_OFFSET+8]$   
 $SRC1\_BYTE2 \leftarrow SRC1[BLK1\_OFFSET+23:BLK1\_OFFSET+16]$   
 $SRC1\_BYTE3 \leftarrow SRC1[BLK1\_OFFSET+31:BLK1\_OFFSET+24]$   
 $SRC1\_BYTE4 \leftarrow SRC1[BLK1\_OFFSET+39:BLK1\_OFFSET+32]$   
 $SRC1\_BYTE5 \leftarrow SRC1[BLK1\_OFFSET+47:BLK1\_OFFSET+40]$   
 $SRC1\_BYTE6 \leftarrow SRC1[BLK1\_OFFSET+55:BLK1\_OFFSET+48]$   
 $SRC1\_BYTE7 \leftarrow SRC1[BLK1\_OFFSET+63:BLK1\_OFFSET+56]$   
 $SRC1\_BYTE8 \leftarrow SRC1[BLK1\_OFFSET+71:BLK1\_OFFSET+64]$   
 $SRC1\_BYTE9 \leftarrow SRC1[BLK1\_OFFSET+79:BLK1\_OFFSET+72]$   
 $SRC1\_BYTE10 \leftarrow SRC1[BLK1\_OFFSET+87:BLK1\_OFFSET+80]$

$SRC2\_BYTE0 \leftarrow SRC2[BLK2\_OFFSET+7:BLK2\_OFFSET]$   
 $SRC2\_BYTE1 \leftarrow SRC2[BLK2\_OFFSET+15:BLK2\_OFFSET+8]$   
 $SRC2\_BYTE2 \leftarrow SRC2[BLK2\_OFFSET+23:BLK2\_OFFSET+16]$   
 $SRC2\_BYTE3 \leftarrow SRC2[BLK2\_OFFSET+31:BLK2\_OFFSET+24]$

$TEMP0 \leftarrow ABS(SRC1\_BYTE0 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE1 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE2 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE3 - SRC2\_BYTE3)$   
 $DEST[143:128] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE1 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE2 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE3 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE4 - SRC2\_BYTE3)$   
 $DEST[159:144] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE2 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE3 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE4 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE5 - SRC2\_BYTE3)$   
 $DEST[175:160] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

TEMP0  $\leftarrow$  ABS(SRC1\_BYTE3 - SRC2\_BYTE0)  
 TEMP1  $\leftarrow$  ABS(SRC1\_BYTE4 - SRC2\_BYTE1)  
 TEMP2  $\leftarrow$  ABS(SRC1\_BYTE5 - SRC2\_BYTE2)  
 TEMP3  $\leftarrow$  ABS(SRC1\_BYTE6 - SRC2\_BYTE3)  
 DEST[191:176]  $\leftarrow$  TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0  $\leftarrow$  ABS(SRC1\_BYTE4 - SRC2\_BYTE0)  
 TEMP1  $\leftarrow$  ABS(SRC1\_BYTE5 - SRC2\_BYTE1)  
 TEMP2  $\leftarrow$  ABS(SRC1\_BYTE6 - SRC2\_BYTE2)  
 TEMP3  $\leftarrow$  ABS(SRC1\_BYTE7 - SRC2\_BYTE3)  
 DEST[207:192]  $\leftarrow$  TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0  $\leftarrow$  ABS(SRC1\_BYTE5 - SRC2\_BYTE0)  
 TEMP1  $\leftarrow$  ABS(SRC1\_BYTE6 - SRC2\_BYTE1)  
 TEMP2  $\leftarrow$  ABS(SRC1\_BYTE7 - SRC2\_BYTE2)  
 TEMP3  $\leftarrow$  ABS(SRC1\_BYTE8 - SRC2\_BYTE3)  
 DEST[223:208]  $\leftarrow$  TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0  $\leftarrow$  ABS(SRC1\_BYTE6 - SRC2\_BYTE0)  
 TEMP1  $\leftarrow$  ABS(SRC1\_BYTE7 - SRC2\_BYTE1)  
 TEMP2  $\leftarrow$  ABS(SRC1\_BYTE8 - SRC2\_BYTE2)  
 TEMP3  $\leftarrow$  ABS(SRC1\_BYTE9 - SRC2\_BYTE3)  
 DEST[239:224]  $\leftarrow$  TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0  $\leftarrow$  ABS(SRC1\_BYTE7 - SRC2\_BYTE0)  
 TEMP1  $\leftarrow$  ABS(SRC1\_BYTE8 - SRC2\_BYTE1)  
 TEMP2  $\leftarrow$  ABS(SRC1\_BYTE9 - SRC2\_BYTE2)  
 TEMP3  $\leftarrow$  ABS(SRC1\_BYTE10 - SRC2\_BYTE3)  
 DEST[255:240]  $\leftarrow$  TEMP0 + TEMP1 + TEMP2 + TEMP3

#### VMPSADBW (VEX.128 encoded version)

BLK2\_OFFSET  $\leftarrow$  imm8[1:0]\*32  
 BLK1\_OFFSET  $\leftarrow$  imm8[2]\*32  
 SRC1\_BYTE0  $\leftarrow$  SRC1[BLK1\_OFFSET+7:BLK1\_OFFSET]  
 SRC1\_BYTE1  $\leftarrow$  SRC1[BLK1\_OFFSET+15:BLK1\_OFFSET+8]  
 SRC1\_BYTE2  $\leftarrow$  SRC1[BLK1\_OFFSET+23:BLK1\_OFFSET+16]  
 SRC1\_BYTE3  $\leftarrow$  SRC1[BLK1\_OFFSET+31:BLK1\_OFFSET+24]  
 SRC1\_BYTE4  $\leftarrow$  SRC1[BLK1\_OFFSET+39:BLK1\_OFFSET+32]  
 SRC1\_BYTE5  $\leftarrow$  SRC1[BLK1\_OFFSET+47:BLK1\_OFFSET+40]  
 SRC1\_BYTE6  $\leftarrow$  SRC1[BLK1\_OFFSET+55:BLK1\_OFFSET+48]  
 SRC1\_BYTE7  $\leftarrow$  SRC1[BLK1\_OFFSET+63:BLK1\_OFFSET+56]  
 SRC1\_BYTE8  $\leftarrow$  SRC1[BLK1\_OFFSET+71:BLK1\_OFFSET+64]  
 SRC1\_BYTE9  $\leftarrow$  SRC1[BLK1\_OFFSET+79:BLK1\_OFFSET+72]  
 SRC1\_BYTE10  $\leftarrow$  SRC1[BLK1\_OFFSET+87:BLK1\_OFFSET+80]

SRC2\_BYTE0  $\leftarrow$  SRC2[BLK2\_OFFSET+7:BLK2\_OFFSET]  
 SRC2\_BYTE1  $\leftarrow$  SRC2[BLK2\_OFFSET+15:BLK2\_OFFSET+8]  
 SRC2\_BYTE2  $\leftarrow$  SRC2[BLK2\_OFFSET+23:BLK2\_OFFSET+16]  
 SRC2\_BYTE3  $\leftarrow$  SRC2[BLK2\_OFFSET+31:BLK2\_OFFSET+24]

$TEMP0 \leftarrow ABS(SRC1\_BYTE0 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE1 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE2 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE3 - SRC2\_BYTE3)$   
 $DEST[15:0] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE1 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE2 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE3 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE4 - SRC2\_BYTE3)$   
 $DEST[31:16] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE2 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE3 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE4 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE5 - SRC2\_BYTE3)$   
 $DEST[47:32] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE3 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE4 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE5 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE6 - SRC2\_BYTE3)$   
 $DEST[63:48] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE4 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE5 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE6 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE7 - SRC2\_BYTE3)$   
 $DEST[79:64] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE5 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE6 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE7 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE8 - SRC2\_BYTE3)$   
 $DEST[95:80] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE6 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE7 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE8 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE9 - SRC2\_BYTE3)$   
 $DEST[111:96] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$

$TEMP0 \leftarrow ABS(SRC1\_BYTE7 - SRC2\_BYTE0)$   
 $TEMP1 \leftarrow ABS(SRC1\_BYTE8 - SRC2\_BYTE1)$   
 $TEMP2 \leftarrow ABS(SRC1\_BYTE9 - SRC2\_BYTE2)$   
 $TEMP3 \leftarrow ABS(SRC1\_BYTE10 - SRC2\_BYTE3)$   
 $DEST[127:112] \leftarrow TEMP0 + TEMP1 + TEMP2 + TEMP3$   
 $DEST[MAXVL-1:128] \leftarrow 0$

**MPSADBW (128-bit Legacy SSE version)**

$$\text{SRC\_OFFSET} \leftarrow \text{imm8}[1:0] * 32$$

$$\text{DEST\_OFFSET} \leftarrow \text{imm8}[2] * 32$$

$$\text{DEST\_BYTE0} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+7:\text{DEST\_OFFSET}]$$

$$\text{DEST\_BYTE1} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+15:\text{DEST\_OFFSET}+8]$$

$$\text{DEST\_BYTE2} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+23:\text{DEST\_OFFSET}+16]$$

$$\text{DEST\_BYTE3} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+31:\text{DEST\_OFFSET}+24]$$

$$\text{DEST\_BYTE4} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+39:\text{DEST\_OFFSET}+32]$$

$$\text{DEST\_BYTE5} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+47:\text{DEST\_OFFSET}+40]$$

$$\text{DEST\_BYTE6} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+55:\text{DEST\_OFFSET}+48]$$

$$\text{DEST\_BYTE7} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+63:\text{DEST\_OFFSET}+56]$$

$$\text{DEST\_BYTE8} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+71:\text{DEST\_OFFSET}+64]$$

$$\text{DEST\_BYTE9} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+79:\text{DEST\_OFFSET}+72]$$

$$\text{DEST\_BYTE10} \leftarrow \text{DEST}[\text{DEST\_OFFSET}+87:\text{DEST\_OFFSET}+80]$$

$$\text{SRC\_BYTE0} \leftarrow \text{SRC}[\text{SRC\_OFFSET}+7:\text{SRC\_OFFSET}]$$

$$\text{SRC\_BYTE1} \leftarrow \text{SRC}[\text{SRC\_OFFSET}+15:\text{SRC\_OFFSET}+8]$$

$$\text{SRC\_BYTE2} \leftarrow \text{SRC}[\text{SRC\_OFFSET}+23:\text{SRC\_OFFSET}+16]$$

$$\text{SRC\_BYTE3} \leftarrow \text{SRC}[\text{SRC\_OFFSET}+31:\text{SRC\_OFFSET}+24]$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{DEST\_BYTE0} - \text{SRC\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{DEST\_BYTE1} - \text{SRC\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{DEST\_BYTE2} - \text{SRC\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{DEST\_BYTE3} - \text{SRC\_BYTE3})$$

$$\text{DEST}[15:0] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{DEST\_BYTE1} - \text{SRC\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{DEST\_BYTE2} - \text{SRC\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{DEST\_BYTE3} - \text{SRC\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{DEST\_BYTE4} - \text{SRC\_BYTE3})$$

$$\text{DEST}[31:16] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{DEST\_BYTE2} - \text{SRC\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{DEST\_BYTE3} - \text{SRC\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{DEST\_BYTE4} - \text{SRC\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{DEST\_BYTE5} - \text{SRC\_BYTE3})$$

$$\text{DEST}[47:32] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{DEST\_BYTE3} - \text{SRC\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{DEST\_BYTE4} - \text{SRC\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{DEST\_BYTE5} - \text{SRC\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{DEST\_BYTE6} - \text{SRC\_BYTE3})$$

$$\text{DEST}[63:48] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

$$\text{TEMP0} \leftarrow \text{ABS}(\text{DEST\_BYTE4} - \text{SRC\_BYTE0})$$

$$\text{TEMP1} \leftarrow \text{ABS}(\text{DEST\_BYTE5} - \text{SRC\_BYTE1})$$

$$\text{TEMP2} \leftarrow \text{ABS}(\text{DEST\_BYTE6} - \text{SRC\_BYTE2})$$

$$\text{TEMP3} \leftarrow \text{ABS}(\text{DEST\_BYTE7} - \text{SRC\_BYTE3})$$

$$\text{DEST}[79:64] \leftarrow \text{TEMP0} + \text{TEMP1} + \text{TEMP2} + \text{TEMP3}$$

```

TEMP0 ← ABS( DEST_BYTE5 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE6 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE7 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE8 - SRC_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

```

```

TEMP0 ← ABS( DEST_BYTE6 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE7 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE8 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE9 - SRC_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

```

```

TEMP0 ← ABS( DEST_BYTE7 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE8 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE9 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE10 - SRC_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
DEST[MAXVL-1:128] (Unmodified)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

(V)MPSADBW:  __m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);
VMPSADBW:    __m256i _mm256_mpsadbw_epu8 (__m256i s1, __m256i s2, const int mask);

```

### Flags Affected

None

### Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.



## MUL—Unsigned Multiply

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /4	MUL <i>r/m8</i>	M	Valid	Valid	Unsigned multiply ( $AX \leftarrow AL * r/m8$ ).
REX + F6 /4	MUL <i>r/m8</i> *	M	Valid	N.E.	Unsigned multiply ( $AX \leftarrow AL * r/m8$ ).
F7 /4	MUL <i>r/m16</i>	M	Valid	Valid	Unsigned multiply ( $DX:AX \leftarrow AX * r/m16$ ).
F7 /4	MUL <i>r/m32</i>	M	Valid	Valid	Unsigned multiply ( $EDX:EAX \leftarrow EAX * r/m32$ ).
REX.W + F7 /4	MUL <i>r/m64</i>	M	Valid	N.E.	Unsigned multiply ( $RDX:RAX \leftarrow RAX * r/m64$ ).

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 4-9.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

**Table 4-9. MUL Results**

Operand Size	Source 1	Source 2	Destination
Byte	AL	<i>r/m8</i>	AX
Word	AX	<i>r/m16</i>	DX:AX
Doubleword	EAX	<i>r/m32</i>	EDX:EAX
Quadword	RAX	<i>r/m64</i>	RDX:RAX

## Operation

```

IF (Byte operation)
    THEN
         $AX \leftarrow AL * SRC;$ 
    ELSE (* Word or doubleword operation *)
        IF OperandSize = 16
            THEN
                 $DX:AX \leftarrow AX * SRC;$ 
            ELSE IF OperandSize = 32
                THEN  $EDX:EAX \leftarrow EAX * SRC;$  FI;
            ELSE (* OperandSize = 64 *)
                 $RDX:RAX \leftarrow RAX * SRC;$ 
        FI;
    FI;
FI;

```

## Flags Affected

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

## Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## MULPD—Multiply Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 59 /r MULPD xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed double-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 59 /r VMULPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed double-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed double-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.NDS.128.66.0F.W1 59 /r VMULPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1.
EVEX.NDS.256.66.0F.W1 59 /r VMULPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1.
EVEX.NDS.512.66.0F.W1 59 /r VMULPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Multiply packed double-precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiply packed double-precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation****VMULPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← SRC1[i+63:i] \* SRC2[63:0]

ELSE

DEST[i+63:i] ← SRC1[i+63:i] \* SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMULPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] \* SRC2[63:0]

DEST[127:64] ← SRC1[127:64] \* SRC2[127:64]

DEST[191:128] ← SRC1[191:128] \* SRC2[191:128]

DEST[255:192] ← SRC1[255:192] \* SRC2[255:192]

DEST[MAXVL-1:256] ← 0;

.

**VMULPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] \* SRC2[63:0]

DEST[127:64] ← SRC1[127:64] \* SRC2[127:64]

DEST[MAXVL-1:128] ← 0

**MULPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] \* SRC[63:0]

DEST[127:64] ← DEST[127:64] \* SRC[127:64]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMULPD __m512d __mm512_mul_pd( __m512d a, __m512d b);
VMULPD __m512d __mm512_mask_mul_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d __mm512_maskz_mul_pd( __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d __mm512_mul_round_pd( __m512d a, __m512d b, int);
VMULPD __m512d __mm512_mask_mul_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m512d __mm512_maskz_mul_round_pd( __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m256d __mm256_mul_pd( __m256d a, __m256d b);
MULPD __m128d __mm_mul_pd( __m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

## MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 59 /r MULPS xmm1, xmm2/m128	A	V/V	SSE	Multiply packed single-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 59 /r VMULPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed single-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.NDS.256.0F.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed single-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.NDS.128.0F.W0 59 /r VMULPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.
EVEX.NDS.256.0F.W0 59 /r VMULPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.
EVEX.NDS.512.0F.W0 59 /r VMULPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F	Multiply packed single-precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiply the packed single-precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

**Operation****VMULPS (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND SRC2 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC1[i+31:i] \* SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC1[i+31:i] \* SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VMULPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] \* SRC2[31:0]

DEST[63:32] ← SRC1[63:32] \* SRC2[63:32]

DEST[95:64] ← SRC1[95:64] \* SRC2[95:64]

DEST[127:96] ← SRC1[127:96] \* SRC2[127:96]

DEST[159:128] ← SRC1[159:128] \* SRC2[159:128]

DEST[191:160] ← SRC1[191:160] \* SRC2[191:160]

DEST[223:192] ← SRC1[223:192] \* SRC2[223:192]

DEST[255:224] ← SRC1[255:224] \* SRC2[255:224].

DEST[MAXVL-1:256] ← 0;

**VMULPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] \* SRC2[31:0]

DEST[63:32] ← SRC1[63:32] \* SRC2[63:32]

DEST[95:64] ← SRC1[95:64] \* SRC2[95:64]

DEST[127:96] ← SRC1[127:96] \* SRC2[127:96]

DEST[MAXVL-1:128] ← 0

**MULPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0] \* SRC2[31:0]

DEST[63:32] ← SRC1[63:32] \* SRC2[63:32]

DEST[95:64] ← SRC1[95:64] \* SRC2[95:64]

DEST[127:96] ← SRC1[127:96] \* SRC2[127:96]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMULPS __m512 _mm512_mul_ps( __m512 a, __m512 b);
VMULPS __m512 _mm512_mask_mul_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_mul_round_ps( __m512 a, __m512 b, int);
VMULPS __m512 _mm512_mask_mul_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMULPS __m512 _mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMULPS __m256 _mm256_mask_mul_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMULPS __m256 _mm256_maskz_mul_ps(__mmask8 k, __m256 a, __m256 b);
VMULPS __m128 _mm_mask_mul_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULPS __m128 _mm_maskz_mul_ps(__mmask8 k, __m128 a, __m128 b);
VMULPS __m256 _mm256_mul_ps( __m256 a, __m256 b);
MULPS __m128 _mm_mul_ps( __m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.



## MULSD—Multiply Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 59 /r MULSD xmm1,xmm2/m64	A	V/V	SSE2	Multiply the low double-precision floating-point value in xmm2/m64 by low double-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.WIG 59 /r VMULSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.
EVEX.NDS.LIG.F2.0F.W1 59 /r VMULSD xmm1 {k1}{z}, xmm2, xmm3/m64 {er}	C	V/V	AVX512F	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies the low double-precision floating-point value in the second source operand by the low double-precision floating-point value in the first source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VMULSD (EVEX encoded version)**

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
            FI
        FI;
ENDFOR
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**VMULSD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**MULSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMULSD __m128d __mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d __mm_maskz_mul_sd( __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d __mm_mul_round_sd( __m128d a, __m128d b, int);
VMULSD __m128d __mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d __mm_maskz_mul_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d __mm_mul_sd (__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## MULSS—Multiply Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 59 /r MULSS xmm1,xmm2/m32	A	V/V	SSE	Multiply the low single-precision floating-point value in xmm2/m32 by the low single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.
EVEX.NDS.LIG.F3.0F.W0 59 /r VMULSS xmm1 {k1}{z}, xmm2, xmm3/m32 {er}	C	V/V	AVX512F	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies the low single-precision floating-point value from the second source operand by the low single-precision floating-point value in the first source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VMULSS (EVEX encoded version)**

```

IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
            FI
        FI;
ENDFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**VMULSS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**MULSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← DEST[31:0] * SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss(__mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss(__m128 a, __m128 b, int);
VMULSS __m128 _mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 _mm_maskz_mul_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)

```

**SIMD Floating-Point Exceptions**

Underflow, Overflow, Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

## MULX — Unsigned Multiply Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.F2.0F38.W0 F6 /r MULX <i>r32a, r32b, r/m32</i>	RVM	V/V	BMI2	Unsigned multiply of <i>r/m32</i> with EDX without affecting arithmetic flags.
VEX.NDD.LZ.F2.0F38.W1 F6 /r MULX <i>r64a, r64b, r/m64</i>	RVM	V/N.E.	BMI2	Unsigned multiply of <i>r/m64</i> with RDX without affecting arithmetic flags.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (w)	ModRM:r/m (r)	RDX/EDX is implied 64/32 bits source

### Description

Performs an unsigned multiplication of the implicit source operand (EDX/RDX) and the specified source operand (the third operand) and stores the low half of the result in the second destination (second operand), the high half of the result in the first destination operand (first operand), without reading or writing the arithmetic flags. This enables efficient programming where the software can interleave add with carry operations and multiplications.

If the first and second operand are identical, it will contain the high half of the multiplication result.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```
// DEST1: ModRM:reg
// DEST2: VEX.vvvv
IF (OperandSize = 32)
    SRC1 ← EDX;
    DEST2 ← (SRC1*SRC2)[31:0];
    DEST1 ← (SRC1*SRC2)[63:32];
ELSE IF (OperandSize = 64)
    SRC1 ← RDX;
    DEST2 ← (SRC1*SRC2)[63:0];
    DEST1 ← (SRC1*SRC2)[127:64];
FI
```

### Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language when possible.

```
unsigned int mulx_u32(unsigned int a, unsigned int b, unsigned int * hi);
```

```
unsigned __int64 mulx_u64(unsigned __int64 a, unsigned __int64 b, unsigned __int64 * hi);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
#UD If VEX.W = 1.

## MWAIT—Monitor Wait

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 C9	MWAIT	Z0	Valid	Valid	A hint that allow the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MWAIT may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32\_MISC\_ENABLE MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. The first processors to implement MWAIT supported only the zero value for EAX and ECX. Later processors allowed setting ECX[0] to enable masked interrupts as break events for MWAIT (see below). Software can use the CPUID instruction to determine the extensions and hints supported by the processor.

### MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

The following cause the processor to exit the implementation-dependent-optimized state: a store to the address range armed by the MONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent-optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent-optimized state either (1) if the interrupt would be delivered to software (e.g., as it would be if HLT had been executed instead of MWAIT); or (2) if ECX[0] = 1. Software can execute MWAIT with ECX[0] = 1 only if CPUID.05H:ECX[bit 1] = 1. (Implementation-specific conditions may result in an interrupt causing the processor to exit the implementation-dependent-optimized state even if interrupts are masked and ECX[0] = 0.)

Following exit from the implementation-dependent-optimized state, control passes to the instruction following the MWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction following the handling of an SMI.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.

## MWAIT for Power Management

MWAIT accepts a hint and optional extension to the processor that it can enter a specified target C state while waiting for an event or a store operation to the address range armed by MONITOR. Support for MWAIT extensions for power management is indicated by CPUID.05H:ECX[bit 0] reporting 1.

EAX and ECX are used to communicate the additional information to the MWAIT instruction, such as the kind of optimized state the processor should enter. ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. Implementation-specific conditions may cause a processor to ignore the hint and enter a different optimized state. Future processor implementations may implement several optimized “waiting” states and will select among those states based on the hint argument. Table 4-10 describes the meaning of ECX and EAX registers for MWAIT extensions.

**Table 4-10. MWAIT Extension Register (ECX)**

Bits	Description
0	Treat interrupts as break events even if masked (e.g., even if EFLAGS.IF=0). May be set only if CPUID.05H:ECX[bit 1] = 1.
31: 1	Reserved

**Table 4-11. MWAIT Hints Register (EAX)**

Bits	Description
3 : 0	Sub C-state within a C-state, indicated by bits [7:4]
7 : 4	Target C-state* Value of 0 means C1; 1 means C2 and so on Value of 01111B means C0  Note: Target C states for MWAIT extensions are processor-specific C-states, not ACPI C-states
31: 8	Reserved

Note that if MWAIT is used to enter any of the C-states that are numerically higher than C1, a store to the address range armed by the MONITOR instruction will cause the processor to exit MWAIT only if the store was originated by other processor agents. A store from non-processor agent might not cause the processor to exit MWAIT in such cases.

For additional details of MWAIT extensions, see Chapter 14, “Power and Thermal Management,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## Operation

(\* MWAIT takes the argument in EAX as a hint extension and is architected to take the argument in ECX as an instruction extension MWAIT EAX, ECX \*)

```
{
  WHILE ( ("Monitor Hardware is in armed state")) {
    implementation_dependent_optimized_state(EAX, ECX); }
  Set the state of Monitor Hardware as triggered;
}
```

## Intel C/C++ Compiler Intrinsic Equivalent

MWAIT:           void \_mm\_mwait(unsigned extensions, unsigned hints)



## Example

MONITOR/MWAIT instruction pair must be coded in the same loop because execution of the MWAIT instruction will trigger the monitor hardware. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. Setting up MONITOR without executing MWAIT has no adverse effects.

Typically the MONITOR/MWAIT pair is used in a sequence, such as:

```
EAX = Logical Address(Trigger)
```

```
ECX = 0 (*Hints *)
```

```
EDX = 0 (* Hints *)
```

```
IF ( !trigger_store_happened) {
    MONITOR EAX, ECX, EDX
    IF ( !trigger_store_happened ) {
        MWAIT EAX, ECX
    }
}
```

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the monitor instruction. Without the second check that triggering store would go un-noticed. Typical usage of MONITOR and MWAIT would have the above code sequence within a loop.

## Numeric Exceptions

None

## Protected Mode Exceptions

```
#GP(0)      If ECX[31:1] ≠ 0.
              If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD          If CPUID.01H:ECX.MONITOR[bit 3] = 0.
              If current privilege level is not 0.
```

## Real Address Mode Exceptions

```
#GP          If ECX[31:1] ≠ 0.
              If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD          If CPUID.01H:ECX.MONITOR[bit 3] = 0.
```

## Virtual 8086 Mode Exceptions

```
#UD          The MWAIT instruction is not recognized in virtual-8086 mode (even if
              CPUID.01H:ECX.MONITOR[bit 3] = 1).
```

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

```
#GP(0)      If RCX[63:1] ≠ 0.
              If RCX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD          If the current privilege level is not 0.
              If CPUID.01H:ECX.MONITOR[bit 3] = 0.
```

## NEG—Two's Complement Negation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /3	NEG <i>r/m8</i>	M	Valid	Valid	Two's complement negate <i>r/m8</i> .
REX + F6 /3	NEG <i>r/m8</i> *	M	Valid	N.E.	Two's complement negate <i>r/m8</i> .
F7 /3	NEG <i>r/m16</i>	M	Valid	Valid	Two's complement negate <i>r/m16</i> .
F7 /3	NEG <i>r/m32</i>	M	Valid	Valid	Two's complement negate <i>r/m32</i> .
REX.W + F7 /3	NEG <i>r/m64</i>	M	Valid	N.E.	Two's complement negate <i>r/m64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m ( <i>r, w</i> )	NA	NA	NA

### Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
IF DEST = 0
    THEN CF ← 0;
    ELSE CF ← 1;
FI;
DEST ← [- (DEST)]
```

### Flags Affected

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## NOP—No Operation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP 90	NOP	Z0	Valid	Valid	One byte no-operation instruction.
NP 0F 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
NP 0F 1F /0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

### Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of “no operation” as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

**Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction**

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

### Flags Affected

None

### Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

## NOT—One's Complement Negation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F6 /2	NOT <i>r/m8</i>	M	Valid	Valid	Reverse each bit of <i>r/m8</i> .
REX + F6 /2	NOT <i>r/m8</i> *	M	Valid	N.E.	Reverse each bit of <i>r/m8</i> .
F7 /2	NOT <i>r/m16</i>	M	Valid	Valid	Reverse each bit of <i>r/m16</i> .
F7 /2	NOT <i>r/m32</i>	M	Valid	Valid	Reverse each bit of <i>r/m32</i> .
REX.W + F7 /2	NOT <i>r/m64</i>	M	Valid	N.E.	Reverse each bit of <i>r/m64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m ( <i>r, w</i> )	NA	NA	NA

### Description

Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← NOT DEST;

### Flags Affected

None

### Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## OR—Logical Inclusive OR

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0C <i>ib</i>	OR AL, <i>imm8</i>	I	Valid	Valid	AL OR <i>imm8</i> .
0D <i>iw</i>	OR AX, <i>imm16</i>	I	Valid	Valid	AX OR <i>imm16</i> .
0D <i>id</i>	OR EAX, <i>imm32</i>	I	Valid	Valid	EAX OR <i>imm32</i> .
REX.W + 0D <i>id</i>	OR RAX, <i>imm32</i>	I	Valid	N.E.	RAX OR <i>imm32</i> (sign-extended).
80 /1 <i>ib</i>	OR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m8</i> OR <i>imm8</i> .
REX + 80 /1 <i>ib</i>	OR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m8</i> OR <i>imm8</i> .
81 /1 <i>iw</i>	OR <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	<i>r/m16</i> OR <i>imm16</i> .
81 /1 <i>id</i>	OR <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	<i>r/m32</i> OR <i>imm32</i> .
REX.W + 81 /1 <i>id</i>	OR <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	<i>r/m64</i> OR <i>imm32</i> (sign-extended).
83 /1 <i>ib</i>	OR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m16</i> OR <i>imm8</i> (sign-extended).
83 /1 <i>ib</i>	OR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m32</i> OR <i>imm8</i> (sign-extended).
REX.W + 83 /1 <i>ib</i>	OR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m64</i> OR <i>imm8</i> (sign-extended).
08 /r	OR <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	<i>r/m8</i> OR <i>r8</i> .
REX + 08 /r	OR <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	<i>r/m8</i> OR <i>r8</i> .
09 /r	OR <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	<i>r/m16</i> OR <i>r16</i> .
09 /r	OR <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	<i>r/m32</i> OR <i>r32</i> .
REX.W + 09 /r	OR <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	<i>r/m64</i> OR <i>r64</i> .
0A /r	OR <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	<i>r8</i> OR <i>r/m8</i> .
REX + 0A /r	OR <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	<i>r8</i> OR <i>r/m8</i> .
0B /r	OR <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	<i>r16</i> OR <i>r/m16</i> .
0B /r	OR <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	<i>r32</i> OR <i>r/m32</i> .
REX.W + 0B /r	OR <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	<i>r64</i> OR <i>r/m64</i> .

## NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

## Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← DEST OR SRC;

### Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

### Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.



## ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 56/r ORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical OR of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F 56 /r VORPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F 56 /r VORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 56 /r VORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 56 /r VORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 56 /r VORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical OR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation****VORPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[63:0]
        ELSE
          DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[i+63:i]
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VORPD (VEX.256 encoded version)**

```

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE OR SRC2[255:192]
DEST[MAXVL-1:256] ← 0

```

**VORPD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[MAXVL-1:128] ← 0

```

**ORPD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE OR SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VORPD __m512d __mm512_or_pd ( __m512d a, __m512d b);
VORPD __m512d __mm512_mask_or_pd ( __m512d s, __mmask8 k, __m512d a, __m512d b);
VORPD __m512d __mm512_maskz_or_pd ( __mmask8 k, __m512d a, __m512d b);
VORPD __m256d __mm256_mask_or_pd ( __m256d s, __mmask8 k, __m256d a, __m256d b);
VORPD __m256d __mm256_maskz_or_pd ( __mmask8 k, __m256d a, __m256d b);
VORPD __m128d __mm_mask_or_pd ( __m128d s, __mmask8 k, __m128d a, __m128d b);
VORPD __m128d __mm_maskz_or_pd ( __mmask8 k, __m128d a, __m128d b);
VORPD __m256d __mm256_or_pd ( __m256d a, __m256d b);
ORPD __m128d __mm_or_pd ( __m128d a, __m128d b);

```

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 56 /r ORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical OR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.OF 56 /r VORPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.OF 56 /r VORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.OF.W0 56 /r VORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.OF.W0 56 /r VORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.OF.W0 56 /r VORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical OR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation****VORPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VORPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]

DEST[159:128] ← SRC1[159:128] BITWISE OR SRC2[159:128]

DEST[191:160] ← SRC1[191:160] BITWISE OR SRC2[191:160]

DEST[223:192] ← SRC1[223:192] BITWISE OR SRC2[223:192]

DEST[255:224] ← SRC1[255:224] BITWISE OR SRC2[255:224].

DEST[MAXVL-1:256] ← 0

**VORPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]

DEST[MAXVL-1:128] ← 0

**ORPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VORPS __m512 __mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 __mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 __mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 __mm256_mask_or_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VORPS __m256 __mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 __mm_mask_or_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 __mm_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
VORPS __m256 __mm256_or_ps (__m256 a, __m256 b);
ORPS __m128 __mm_or_ps (__m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## OUT—Output to Port

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E6 <i>ib</i>	OUT <i>imm8</i> , AL	I	Valid	Valid	Output byte in AL to I/O port address <i>imm8</i> .
E7 <i>ib</i>	OUT <i>imm8</i> , AX	I	Valid	Valid	Output word in AX to I/O port address <i>imm8</i> .
E7 <i>ib</i>	OUT <i>imm8</i> , EAX	I	Valid	Valid	Output doubleword in EAX to I/O port address <i>imm8</i> .
EE	OUT DX, AL	ZO	Valid	Valid	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	ZO	Valid	Valid	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	ZO	Valid	Valid	Output doubleword in EAX to I/O port address in DX.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	<i>imm8</i>	NA	NA	NA
ZO	NA	NA	NA	NA

### Description

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium® processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

**Operation**

```

IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
  THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
    IF (Any I/O Permission Bit for I/O port being accessed = 1)
      THEN (* I/O operation is not allowed *)
        #GP(0);
      ELSE (* I/O operation is allowed *)
        DEST ← SRC; (* Writes to selected I/O port *)
    FI;
  ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST ← SRC; (* Writes to selected I/O port *)
  FI;

```

**Flags Affected**

None

**Protected Mode Exceptions**

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD	If the LOCK prefix is used.
-----	-----------------------------

**Virtual-8086 Mode Exceptions**

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same as protected mode exceptions.

**64-Bit Mode Exceptions**

Same as protected mode exceptions.



## OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
6E	OUTS DX, <i>m8</i>	ZO	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, <i>m16</i>	ZO	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, <i>m32</i>	ZO	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6E	OUTSB	ZO	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSW	ZO	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSD	ZO	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.

## NOTES:

\* See IA-32 Architecture Compatibility section below.

\*\* In 64-bit mode, only 64-bit (RSI) and 32-bit (ESI) address sizes are supported. In non-64-bit mode, only 32-bit (ESI) and 16-bit (SI) address sizes are supported.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

## Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:SI, DS:ESI or the RSI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The DS segment may be overridden with a segment override prefix.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the OUTS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand should be a symbol that indicates the size of the I/O port and the source address, and the destination operand must be DX. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI or RSI registers, which must be loaded correctly before the OUTS instruction is executed.

The no-operands form provides “short forms” of the byte, word, and doubleword versions of the OUTS instructions. Here also DS:(E)SI is assumed to be the source operand and DX is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: OUTSB (byte), OUTSW (word), or OUTSD (doubleword).

After the byte, word, or doubleword is transferred from the memory location to the I/O port, the SI/ESI/RSI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the SI/ESI/RSI register is decremented.) The SI/ESI/RSI register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix. This instruction is only useful for accessing I/O ports located in the processor’s I/O address space. See Chapter 18, “Input/Output,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, the default operand size is 32 bits; operand size is not promoted by the use of REX.W. In 64-bit mode, the default address size is 64 bits, and 64-bit address is specified using RSI by default. 32-bit address using ESI is support using the prefix 67H, but 16-bit address is not supported in 64-bit mode.

## IA-32 Architecture Compatibility

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

For the Pentium 4, Intel® Xeon®, and P6 processor family, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

## Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST ← SRC; (* Writes to I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode or 64-Bit Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Writes to I/O port *)
    FI;
```

Byte transfer:

```
IF 64-bit mode
    Then
        IF 64-Bit Address Size
            THEN
                IF DF = 0
                    THEN RSI ← RSI + 1;
                    ELSE RSI ← RSI - 1;
                FI;
            ELSE (* 32-Bit Address Size *)
                IF DF = 0
                    THEN ESI ← ESI + 1;
                    ELSE ESI ← ESI - 1;
                FI;
        FI;
    ELSE
        IF DF = 0
            THEN (ESI) ← (ESI) + 1;
            ELSE (ESI) ← (ESI) - 1;
        FI;
```

Word transfer:

```
IF 64-bit mode
```

```

Then
  IF 64-Bit Address Size
    THEN
      IF DF = 0
        THEN RSI  $\leftarrow$  RSI + 2;
        ELSE RSI  $\leftarrow$  RSI - 2;
      FI;
    ELSE (* 32-Bit Address Size *)
      IF DF = 0
        THEN ESI  $\leftarrow$  ESI + 2;
        ELSE ESI  $\leftarrow$  ESI - 2;
      FI;
    FI;
  ELSE
    IF DF = 0
      THEN (ESI)  $\leftarrow$  (ESI) + 2;
      ELSE (ESI)  $\leftarrow$  (ESI) - 2;
    FI;
  FI;
Doubleword transfer:
IF 64-bit mode
  Then
    IF 64-Bit Address Size
      THEN
        IF DF = 0
          THEN RSI  $\leftarrow$  RSI + 4;
          ELSE RSI  $\leftarrow$  RSI - 4;
        FI;
      ELSE (* 32-Bit Address Size *)
        IF DF = 0
          THEN ESI  $\leftarrow$  ESI + 4;
          ELSE ESI  $\leftarrow$  ESI - 4;
        FI;
      FI;
    ELSE
      IF DF = 0
        THEN (ESI)  $\leftarrow$  (ESI) + 4;
        ELSE (ESI)  $\leftarrow$  (ESI) - 4;
      FI;
    FI;
  FI;

```

**Flags Affected**

None

**Protected Mode Exceptions**

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. If a memory operand effective address is outside the limit of the CS, DS, ES, FS, or GS segment. If the segment register contains a NULL segment selector.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same as for protected mode exceptions.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1. If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**PABSB/PABSW/PABSD/PABSQ — Packed Absolute Value**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 1C /r <sup>1</sup> PABSB <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSSE3	Compute the absolute value of bytes in <i>mm2/m64</i> and store UNSIGNED result in <i>mm1</i> .
66 0F 38 1C /r PABSB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSSE3	Compute the absolute value of bytes in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
NP 0F 38 1D /r <sup>1</sup> PABSW <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in <i>mm2/m64</i> and store UNSIGNED result in <i>mm1</i> .
66 0F 38 1D /r PABSW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
NP 0F 38 1E /r <sup>1</sup> PABSD <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in <i>mm2/m64</i> and store UNSIGNED result in <i>mm1</i> .
66 0F 38 1E /r PABSD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.128.66.0F38.WIG 1C /r VPABSB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	AVX	Compute the absolute value of bytes in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.128.66.0F38.WIG 1D /r VPABSW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	AVX	Compute the absolute value of 16-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.128.66.0F38.WIG 1E /r VPABSD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	AVX	Compute the absolute value of 32-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> .
VEX.256.66.0F38.WIG 1C /r VPABSB <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	AVX2	Compute the absolute value of bytes in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> .
VEX.256.66.0F38.WIG 1D /r VPABSW <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	AVX2	Compute the absolute value of 16-bit integers in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> .
VEX.256.66.0F38.WIG 1E /r VPABSD <i>ymm1</i> , <i>ymm2/m256</i>	A	V/V	AVX2	Compute the absolute value of 32-bit integers in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> .
EVEX.128.66.0F38.WIG 1C /r VPABSB <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2/m128</i>	B	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.256.66.0F38.WIG 1C /r VPABSB <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2/m256</i>	B	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in <i>ymm2/m256</i> and store UNSIGNED result in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.512.66.0F38.WIG 1C /r VPABSB <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2/m512</i>	B	V/V	AVX512BW	Compute the absolute value of bytes in <i>zmm2/m512</i> and store UNSIGNED result in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.128.66.0F38.WIG 1D /r VPABSW <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2/m128</i>	B	V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in <i>xmm2/m128</i> and store UNSIGNED result in <i>xmm1</i> using writemask <i>k1</i> .

EVEX.256.66.0F38.WIG 1D /r VPABSW ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1D /r VPABSW zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW	Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 1E /r VPABSD xmm1 {k1}{z}, xmm2/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in xmm2/m128/m32bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 1E /r VPABSD ymm1 {k1}{z}, ymm2/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in ymm2/m256/m32bcst and store UNSIGNED result in ymm1 using writemask k1.
VPABSD zmm1 {k1}{z}, zmm2/m512/m32bcst	C	V/V	AVX512F	Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 1F /r VPABSQ xmm1 {k1}{z}, xmm2/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in xmm2/m128/m64bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 1F /r VPABSQ ymm1 {k1}{z}, ymm2/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in ymm2/m256/m64bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 1F /r VPABSQ zmm1 {k1}{z}, zmm2/m512/m64bcst	C	V/V	AVX512F	Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded versions: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded versions: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL\_MAX-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### PABSB with 128 bit operands:

```
Unsigned DEST[7:0] ← ABS(SRC[7:0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] ← ABS(SRC[127:120])
```

#### VPABSB with 128 bit operands:

```
Unsigned DEST[7:0] ← ABS(SRC[7:0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] ← ABS(SRC[127:120])
```

#### VPABSB with 256 bit operands:

```
Unsigned DEST[7:0] ← ABS(SRC[7:0])
Repeat operation for 2nd through 31st bytes
Unsigned DEST[255:248] ← ABS(SRC[255:248])
```

#### VPABSB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN
      Unsigned DEST[i+7:i] ← ABS(SRC[i+7:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] ← 0
      FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

#### PABSW with 128 bit operands:

```
Unsigned DEST[15:0] ← ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127:112] ← ABS(SRC[127:112])
```

#### VPABSW with 128 bit operands:

```
Unsigned DEST[15:0] ← ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127:112] ← ABS(SRC[127:112])
```

#### VPABSW with 256 bit operands:

```
Unsigned DEST[15:0] ← ABS(SRC[15:0])
Repeat operation for 2nd through 15th 16-bit words
Unsigned DEST[255:240] ← ABS(SRC[255:240])
```

**VPABSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      Unsigned DEST[j+15:i] ← ABS(SRC[j+15:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[j+15:i] ← 0
      FI
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**PABSD with 128 bit operands:**

```

Unsigned DEST[31:0] ← ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96] ← ABS(SRC[127:96])

```

**VPABSD with 128 bit operands:**

```

Unsigned DEST[31:0] ← ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96] ← ABS(SRC[127:96])

```

**VPABSD with 256 bit operands:**

```

Unsigned DEST[31:0] ← ABS(SRC[31:0])
Repeat operation for 2nd through 7th 32-bit double words
Unsigned DEST[255:224] ← ABS(SRC[255:224])

```

**VPABSD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN
          Unsigned DEST[j+31:i] ← ABS(SRC[31:0])
        ELSE
          Unsigned DEST[j+31:i] ← ABS(SRC[j+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[j+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[j+31:i] ← 0
        FI
      FI;
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```



**VPABSQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN

Unsigned DEST[i+63:i] ← ABS(SRC[63:0])

ELSE

Unsigned DEST[i+63:i] ← ABS(SRC[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPABSB \_\_m512i \_\_mm512\_abs\_epi8 ( \_\_m512i a)

VPABSW \_\_m512i \_\_mm512\_abs\_epi16 ( \_\_m512i a)

VPABSB \_\_m512i \_\_mm512\_mask\_abs\_epi8 ( \_\_m512i s, \_\_mmask64 m, \_\_m512i a)

VPABSW \_\_m512i \_\_mm512\_mask\_abs\_epi16 ( \_\_m512i s, \_\_mmask32 m, \_\_m512i a)

VPABSB \_\_m512i \_\_mm512\_maskz\_abs\_epi8 ( \_\_mmask64 m, \_\_m512i a)

VPABSW \_\_m512i \_\_mm512\_maskz\_abs\_epi16 ( \_\_mmask32 m, \_\_m512i a)

VPABSB \_\_m256i \_\_mm256\_mask\_abs\_epi8 ( \_\_m256i s, \_\_mmask32 m, \_\_m256i a)

VPABSW \_\_m256i \_\_mm256\_mask\_abs\_epi16 ( \_\_m256i s, \_\_mmask16 m, \_\_m256i a)

VPABSB \_\_m256i \_\_mm256\_maskz\_abs\_epi8 ( \_\_mmask32 m, \_\_m256i a)

VPABSW \_\_m256i \_\_mm256\_maskz\_abs\_epi16 ( \_\_mmask16 m, \_\_m256i a)

VPABSB \_\_m128i \_\_mm\_mask\_abs\_epi8 ( \_\_m128i s, \_\_mmask16 m, \_\_m128i a)

VPABSW \_\_m128i \_\_mm\_mask\_abs\_epi16 ( \_\_m128i s, \_\_mmask8 m, \_\_m128i a)

VPABSB \_\_m128i \_\_mm\_maskz\_abs\_epi8 ( \_\_mmask16 m, \_\_m128i a)

VPABSW \_\_m128i \_\_mm\_maskz\_abs\_epi16 ( \_\_mmask8 m, \_\_m128i a)

VPABSD \_\_m256i \_\_mm256\_mask\_abs\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256i a);

VPABSD \_\_m256i \_\_mm256\_maskz\_abs\_epi32( \_\_mmask8 k, \_\_m256i a);

VPABSD \_\_m128i \_\_mm\_mask\_abs\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a);

VPABSD \_\_m128i \_\_mm\_maskz\_abs\_epi32( \_\_mmask8 k, \_\_m128i a);

VPABSD \_\_m512i \_\_mm512\_abs\_epi32( \_\_m512i a);

VPABSD \_\_m512i \_\_mm512\_mask\_abs\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a);

VPABSD \_\_m512i \_\_mm512\_maskz\_abs\_epi32( \_\_mmask16 k, \_\_m512i a);

VPABSQ \_\_m512i \_\_mm512\_abs\_epi64( \_\_m512i a);

VPABSQ \_\_m512i \_\_mm512\_mask\_abs\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a);

VPABSQ \_\_m512i \_\_mm512\_maskz\_abs\_epi64( \_\_mmask8 k, \_\_m512i a);

VPABSQ \_\_m256i \_\_mm256\_mask\_abs\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a);

VPABSQ \_\_m256i \_\_mm256\_maskz\_abs\_epi64( \_\_mmask8 k, \_\_m256i a);

VPABSQ \_\_m128i \_\_mm\_mask\_abs\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a);

VPABSQ \_\_m128i \_\_mm\_maskz\_abs\_epi64( \_\_mmask8 k, \_\_m128i a);

PABSB \_\_m128i \_\_mm\_abs\_epi8 ( \_\_m128i a)

VPABSB \_\_m128i \_\_mm\_abs\_epi8 ( \_\_m128i a)

VPABSB \_\_m256i \_\_mm256\_abs\_epi8 (\_\_m256i a)  
 PABSW \_\_m128i \_\_mm\_abs\_epi16 (\_\_m128i a)  
 VPABSW \_\_m128i \_\_mm\_abs\_epi16 (\_\_m128i a)  
 VPABSW \_\_m256i \_\_mm256\_abs\_epi16 (\_\_m256i a)  
 PABSD \_\_m128i \_\_mm\_abs\_epi32 (\_\_m128i a)  
 VPABSD \_\_m128i \_\_mm\_abs\_epi32 (\_\_m128i a)  
 VPABSD \_\_m256i \_\_mm256\_abs\_epi32 (\_\_m256i a)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPABSD/Q, see Exceptions Type E4.

EVEX-encoded VPABSB/W, see Exceptions Type E4.nb.

## PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 63 /r <sup>1</sup> PACKSSWB <i>mm1</i> , <i>mm2/m64</i>	A	V/V	MMX	Converts 4 packed signed word integers from <i>mm1</i> and from <i>mm2/m64</i> into 8 packed signed byte integers in <i>mm1</i> using signed saturation.
66 0F 63 /r PACKSSWB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Converts 8 packed signed word integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
NP 0F 6B /r <sup>1</sup> PACKSSDW <i>mm1</i> , <i>mm2/m64</i>	A	V/V	MMX	Converts 2 packed signed doubleword integers from <i>mm1</i> and from <i>mm2/m64</i> into 4 packed signed word integers in <i>mm1</i> using signed saturation.
66 0F 6B /r PACKSSDW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Converts 4 packed signed doubleword integers from <i>xmm1</i> and from <i>xmm2/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Converts 8 packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 16 packed signed byte integers in <i>xmm1</i> using signed saturation.
VEX.NDS.128.66.0F.WIG 6B /r VPACKSSDW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Converts 4 packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128</i> into 8 packed signed word integers in <i>xmm1</i> using signed saturation.
VEX.NDS.256.66.0F.WIG 63 /r VPACKSSWB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Converts 16 packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 32 packed signed byte integers in <i>ymm1</i> using signed saturation.
VEX.NDS.256.66.0F.WIG 6B /r VPACKSSDW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Converts 8 packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256</i> into 16 packed signed word integers in <i>ymm1</i> using signed saturation.
EVEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Converts packed signed word integers from <i>xmm2</i> and from <i>xmm3/m128</i> into packed signed byte integers in <i>xmm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG 63 /r VPACKSSWB <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Converts packed signed word integers from <i>ymm2</i> and from <i>ymm3/m256</i> into packed signed byte integers in <i>ymm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG 63 /r VPACKSSWB <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Converts packed signed word integers from <i>zmm2</i> and from <i>zmm3/m512</i> into packed signed byte integers in <i>zmm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W0 6B /r VPACKSSDW <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	D	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from <i>xmm2</i> and from <i>xmm3/m128/m32bcst</i> into packed signed word integers in <i>xmm1</i> using signed saturation under writemask <i>k1</i> .

EVEX.NDS.256.66.0F.W0 6B /r VPACKSSDW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from <i>ymm2</i> and from <i>ymm3/m256/m32bcst</i> into packed signed word integers in <i>ymm1</i> using signed saturation under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W0 6B /r VPACKSSDW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512BW	Converts packed signed doubleword integers from <i>zmm2</i> and from <i>zmm3/m512/m32bcst</i> into packed signed word integers in <i>zmm1</i> using signed saturation under writemask <i>k1</i> .

**NOTES:**

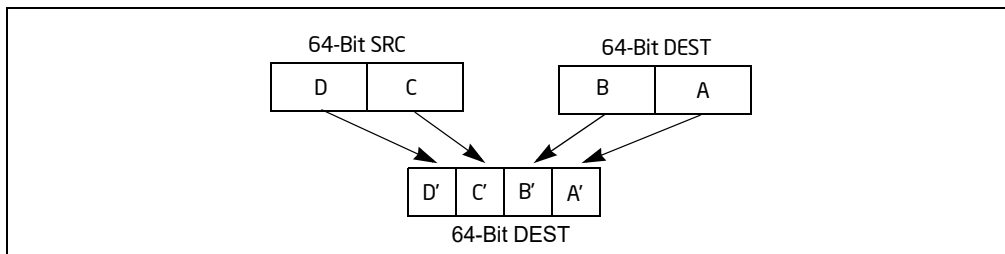
1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-6 for an example of the packing operation.



**Figure 4-6. Operation of the PACKSSDW Instruction Using 64-bit Operands**

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed doubleword value is beyond the range of an unsigned word (i.e. greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask *k1*.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask *k1*.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM destination register destination are unmodified.

## Operation

### PACKSSWB instruction (128-bit Legacy SSE version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified)
```

### PACKSSDW instruction (128-bit Legacy SSE version)

```
DEST[15:0] ← SaturateSignedDwordToSignedWord (DEST[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (DEST[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (DEST[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (DEST[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC[127:96]);
DEST[MAXVL-1:128] (Unmodified)
```

**VPACKSSWB instruction (VEX.128 encoded version)**

DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);  
 DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);  
 DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);  
 DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);  
 DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);  
 DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);  
 DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);  
 DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);  
 DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);  
 DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);  
 DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);  
 DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);  
 DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);  
 DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);  
 DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);  
 DEST[MAXVL-1:128] ← 0;

**VPACKSSDW instruction (VEX.128 encoded version)**

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);  
 DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);  
 DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);  
 DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);  
 DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);  
 DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);  
 DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);  
 DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);  
 DEST[MAXVL-1:128] ← 0;

**VPACKSSWB instruction (VEX.256 encoded version)**

DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);  
 DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);  
 DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);  
 DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);  
 DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);  
 DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);  
 DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);  
 DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);  
 DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);  
 DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);  
 DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);  
 DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);  
 DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);  
 DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);  
 DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);  
 DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);  
 DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);  
 DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);  
 DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);  
 DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);  
 DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);  
 DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);

DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);  
 DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);  
 DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);  
 DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);  
 DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);  
 DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);  
 DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);  
 DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);  
 DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);  
 DEST[MAXVL-1:256] ← 0;

#### VPACKSSDW instruction (VEX.256 encoded version)

DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);  
 DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);  
 DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);  
 DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);  
 DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);  
 DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);  
 DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);  
 DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);  
 DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);  
 DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);  
 DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);  
 DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);  
 DEST[207:192] ← SaturateSignedDwordToSignedWord (SRC2[159:128]);  
 DEST[223:208] ← SaturateSignedDwordToSignedWord (SRC2[191:160]);  
 DEST[239:224] ← SaturateSignedDwordToSignedWord (SRC2[223:192]);  
 DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);  
 DEST[MAXVL-1:256] ← 0;

#### VPACKSSWB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

TMP\_DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);  
 TMP\_DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);  
 TMP\_DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);  
 TMP\_DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);  
 TMP\_DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);  
 TMP\_DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);  
 TMP\_DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);  
 TMP\_DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);  
 TMP\_DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);  
 TMP\_DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);  
 TMP\_DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);  
 TMP\_DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);  
 TMP\_DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);  
 TMP\_DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);  
 TMP\_DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);  
 TMP\_DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);  
 IF VL >= 256  
   TMP\_DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);  
   TMP\_DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);  
   TMP\_DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);  
   TMP\_DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);  
   TMP\_DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);



```

TMP_DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
TMP_DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
TMP_DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
TMP_DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
TMP_DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
TMP_DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
TMP_DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
TMP_DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
TMP_DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
TMP_DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
TMP_DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);

```

```

FI;

```

```

IF VL >= 512

```

```

    TMP_DEST[263:256] ← SaturateSignedWordToSignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToSignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToSignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToSignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToSignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToSignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToSignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToSignedByte (SRC1[383:368]);

```

```

    TMP_DEST[327:320] ← SaturateSignedWordToSignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToSignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToSignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToSignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToSignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToSignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToSignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToSignedByte (SRC2[383:368]);

```

```

    TMP_DEST[391:384] ← SaturateSignedWordToSignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToSignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToSignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToSignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToSignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToSignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToSignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToSignedByte (SRC1[511:496]);

```

```

    TMP_DEST[455:448] ← SaturateSignedWordToSignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToSignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToSignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToSignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToSignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToSignedByte (SRC2[479:464]);
    TMP_DEST[503:496] ← SaturateSignedWordToSignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToSignedByte (SRC2[511:496]);

```

```

FI;

```

```

FOR j ← 0 TO KL-1

```

```

    i ← j * 8

```

```

    IF k1[j] OR *no writemask*

```

```

        THEN

```

```

            DEST[i+7:i] ← TMP_DEST[i+7:i]

```



```

ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[j+7:i] remains unchanged*
    ELSE *zeroing-masking*                ; zeroing-masking
        DEST[j+7:i] ← 0
    FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

VPACKSSDW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO ((KL/2) - 1)
    i ← j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[j+31:i] ← SRC2[31:0]
        ELSE
            TMP_SRC2[j+31:i] ← SRC2[j+31:i]
        FI;
    ENDFOR;

    TMP_DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
    TMP_DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
    TMP_DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
    TMP_DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
    TMP_DEST[79:64] ← SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
    TMP_DEST[95:80] ← SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
    TMP_DEST[111:96] ← SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
    TMP_DEST[127:112] ← SaturateSignedDwordToSignedWord (TMP_SRC2[127:96]);
    IF VL >= 256
        TMP_DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
        TMP_DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
        TMP_DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
        TMP_DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
        TMP_DEST[207:192] ← SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
        TMP_DEST[223:208] ← SaturateSignedDwordToSignedWord (TMP_SRC2[191:160]);
        TMP_DEST[239:224] ← SaturateSignedDwordToSignedWord (TMP_SRC2[223:192]);
        TMP_DEST[255:240] ← SaturateSignedDwordToSignedWord (TMP_SRC2[255:224]);
    FI;
    IF VL >= 512
        TMP_DEST[271:256] ← SaturateSignedDwordToSignedWord (SRC1[287:256]);
        TMP_DEST[287:272] ← SaturateSignedDwordToSignedWord (SRC1[319:288]);
        TMP_DEST[303:288] ← SaturateSignedDwordToSignedWord (SRC1[351:320]);
        TMP_DEST[319:304] ← SaturateSignedDwordToSignedWord (SRC1[383:352]);
        TMP_DEST[335:320] ← SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
        TMP_DEST[351:336] ← SaturateSignedDwordToSignedWord (TMP_SRC2[319:288]);
        TMP_DEST[367:352] ← SaturateSignedDwordToSignedWord (TMP_SRC2[351:320]);
        TMP_DEST[383:368] ← SaturateSignedDwordToSignedWord (TMP_SRC2[383:352]);

        TMP_DEST[399:384] ← SaturateSignedDwordToSignedWord (SRC1[415:384]);
        TMP_DEST[415:400] ← SaturateSignedDwordToSignedWord (SRC1[447:416]);
        TMP_DEST[431:416] ← SaturateSignedDwordToSignedWord (SRC1[479:448]);
    FI;

```

```

TMP_DEST[447:432] ← SaturateSignedDwordToSignedWord (SRC1[511:480]);
TMP_DEST[463:448] ← SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
TMP_DEST[479:464] ← SaturateSignedDwordToSignedWord (TMP_SRC2[447:416]);
TMP_DEST[495:480] ← SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
TMP_DEST[511:496] ← SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] ← 0
    FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPACKSSDW __m512i __mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW __m512i __mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW __m512i __mm512_maskz_packs_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW __m256i __mm256_mask_packs_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW __m256i __mm256_maskz_packs_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW __m128i __mm_mask_packs_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW __m128i __mm_maskz_packs_epi32(__mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB __m512i __mm512_packs_epi16(__m512i m1, __m512i m2);
VPACKSSWB __m512i __mm512_mask_packs_epi16(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB __m512i __mm512_maskz_packs_epi16(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB __m256i __mm256_mask_packs_epi16(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB __m256i __mm256_maskz_packs_epi16(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB __m128i __mm_mask_packs_epi16(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB __m128i __mm_maskz_packs_epi16(__mmask8 k, __m128i m1, __m128i m2);
PACKSSWB __m128i __mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSDW __m128i __mm_packs_epi32(__m128i m1, __m128i m2)
VPACKSSWB __m256i __mm256_packs_epi16(__m256i m1, __m256i m2)
VPACKSSDW __m256i __mm256_packs_epi32(__m256i m1, __m256i m2)

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPACKSSDW, see Exceptions Type E4NF.

EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb.

## PACKUSDW—Pack with Unsigned Saturation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW <i>xmm1, xmm2/m128</i>	A	V/V	SSE4_1	Convert 4 packed signed doubleword integers from <i>xmm1</i> and 4 packed signed doubleword integers from <i>xmm2/m128</i> into 8 packed unsigned word integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.128.66.0F38 2B /r VPACKUSDW <i>xmm1, xmm2, xmm3/m128</i>	B	V/V	AVX	Convert 4 packed signed doubleword integers from <i>xmm2</i> and 4 packed signed doubleword integers from <i>xmm3/m128</i> into 8 packed unsigned word integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.256.66.0F38 2B /r VPACKUSDW <i>ymm1, ymm2, ymm3/m256</i>	B	V/V	AVX2	Convert 8 packed signed doubleword integers from <i>ymm2</i> and 8 packed signed doubleword integers from <i>ymm3/m256</i> into 16 packed unsigned word integers in <i>ymm1</i> using unsigned saturation.
EVEX.NDS.128.66.0F38.W0 2B /r VPACKUSDW <i>xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst</i>	C	V/V	AVX512VL AVX512BW	Convert packed signed doubleword integers from <i>xmm2</i> and packed signed doubleword integers from <i>xmm3/m128/m32bcst</i> into packed unsigned word integers in <i>xmm1</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.NDS.256.66.0F38.W0 2B /r	C	V/V	AVX512VL AVX512BW	Convert packed signed doubleword integers from <i>ymm2</i> and packed signed doubleword integers from <i>ymm3/m256/m32bcst</i> into packed unsigned word integers in <i>ymm1</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.NDS.512.66.0F38.W0 2B /r VPACKUSDW <i>zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst</i>	C	V/V	AVX512BW	Convert packed signed doubleword integers from <i>zmm2</i> and packed signed doubleword integers from <i>zmm3/m512/m32bcst</i> into packed unsigned word integers in <i>zmm1</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, updated conditionally under the writemask *k1*.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding destination register destination are unmodified.

**Operation****PACKUSDW (Legacy SSE instruction)**

$\text{TMP}[15:0] \leftarrow (\text{DEST}[31:0] < 0) ? 0 : \text{DEST}[15:0];$   
 $\text{DEST}[15:0] \leftarrow (\text{DEST}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[15:0];$   
 $\text{TMP}[31:16] \leftarrow (\text{DEST}[63:32] < 0) ? 0 : \text{DEST}[47:32];$   
 $\text{DEST}[31:16] \leftarrow (\text{DEST}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[31:16];$   
 $\text{TMP}[47:32] \leftarrow (\text{DEST}[95:64] < 0) ? 0 : \text{DEST}[79:64];$   
 $\text{DEST}[47:32] \leftarrow (\text{DEST}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[47:32];$   
 $\text{TMP}[63:48] \leftarrow (\text{DEST}[127:96] < 0) ? 0 : \text{DEST}[111:96];$   
 $\text{DEST}[63:48] \leftarrow (\text{DEST}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[63:48];$   
 $\text{TMP}[79:64] \leftarrow (\text{SRC}[31:0] < 0) ? 0 : \text{SRC}[15:0];$   
 $\text{DEST}[79:64] \leftarrow (\text{SRC}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[79:64];$   
 $\text{TMP}[95:80] \leftarrow (\text{SRC}[63:32] < 0) ? 0 : \text{SRC}[47:32];$   
 $\text{DEST}[95:80] \leftarrow (\text{SRC}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[95:80];$   
 $\text{TMP}[111:96] \leftarrow (\text{SRC}[95:64] < 0) ? 0 : \text{SRC}[79:64];$   
 $\text{DEST}[111:96] \leftarrow (\text{SRC}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[111:96];$   
 $\text{TMP}[127:112] \leftarrow (\text{SRC}[127:96] < 0) ? 0 : \text{SRC}[111:96];$   
 $\text{DEST}[127:112] \leftarrow (\text{SRC}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[127:112];$   
 $\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}$

**PACKUSDW (VEX.128 encoded version)**

$\text{TMP}[15:0] \leftarrow (\text{SRC1}[31:0] < 0) ? 0 : \text{SRC1}[15:0];$   
 $\text{DEST}[15:0] \leftarrow (\text{SRC1}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[15:0];$   
 $\text{TMP}[31:16] \leftarrow (\text{SRC1}[63:32] < 0) ? 0 : \text{SRC1}[47:32];$   
 $\text{DEST}[31:16] \leftarrow (\text{SRC1}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[31:16];$   
 $\text{TMP}[47:32] \leftarrow (\text{SRC1}[95:64] < 0) ? 0 : \text{SRC1}[79:64];$   
 $\text{DEST}[47:32] \leftarrow (\text{SRC1}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[47:32];$   
 $\text{TMP}[63:48] \leftarrow (\text{SRC1}[127:96] < 0) ? 0 : \text{SRC1}[111:96];$   
 $\text{DEST}[63:48] \leftarrow (\text{SRC1}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[63:48];$   
 $\text{TMP}[79:64] \leftarrow (\text{SRC2}[31:0] < 0) ? 0 : \text{SRC2}[15:0];$   
 $\text{DEST}[79:64] \leftarrow (\text{SRC2}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[79:64];$   
 $\text{TMP}[95:80] \leftarrow (\text{SRC2}[63:32] < 0) ? 0 : \text{SRC2}[47:32];$   
 $\text{DEST}[95:80] \leftarrow (\text{SRC2}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[95:80];$   
 $\text{TMP}[111:96] \leftarrow (\text{SRC2}[95:64] < 0) ? 0 : \text{SRC2}[79:64];$   
 $\text{DEST}[111:96] \leftarrow (\text{SRC2}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[111:96];$   
 $\text{TMP}[127:112] \leftarrow (\text{SRC2}[127:96] < 0) ? 0 : \text{SRC2}[111:96];$   
 $\text{DEST}[127:112] \leftarrow (\text{SRC2}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[127:112];$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0;$

**VPACKUSDW (VEX.256 encoded version)**

$\text{TMP}[15:0] \leftarrow (\text{SRC1}[31:0] < 0) ? 0 : \text{SRC1}[15:0];$   
 $\text{DEST}[15:0] \leftarrow (\text{SRC1}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[15:0];$   
 $\text{TMP}[31:16] \leftarrow (\text{SRC1}[63:32] < 0) ? 0 : \text{SRC1}[47:32];$   
 $\text{DEST}[31:16] \leftarrow (\text{SRC1}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[31:16];$   
 $\text{TMP}[47:32] \leftarrow (\text{SRC1}[95:64] < 0) ? 0 : \text{SRC1}[79:64];$   
 $\text{DEST}[47:32] \leftarrow (\text{SRC1}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[47:32];$   
 $\text{TMP}[63:48] \leftarrow (\text{SRC1}[127:96] < 0) ? 0 : \text{SRC1}[111:96];$   
 $\text{DEST}[63:48] \leftarrow (\text{SRC1}[127:96] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[63:48];$   
 $\text{TMP}[79:64] \leftarrow (\text{SRC2}[31:0] < 0) ? 0 : \text{SRC2}[15:0];$   
 $\text{DEST}[79:64] \leftarrow (\text{SRC2}[31:0] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[79:64];$   
 $\text{TMP}[95:80] \leftarrow (\text{SRC2}[63:32] < 0) ? 0 : \text{SRC2}[47:32];$   
 $\text{DEST}[95:80] \leftarrow (\text{SRC2}[63:32] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[95:80];$   
 $\text{TMP}[111:96] \leftarrow (\text{SRC2}[95:64] < 0) ? 0 : \text{SRC2}[79:64];$   
 $\text{DEST}[111:96] \leftarrow (\text{SRC2}[95:64] > \text{FFFFH}) ? \text{FFFFH} : \text{TMP}[111:96];$

```

TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128];
TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144];
TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160];
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176];
TMP[207:192] ← (SRC2[159:128] < 0) ? 0 : SRC2[143:128];
DEST[207:192] ← (SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192];
TMP[223:208] ← (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] ← (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208];
TMP[239:224] ← (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] ← (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224];
TMP[255:240] ← (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] ← (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240];
DEST[MAXVL-1:256] ← 0;

```

**VPACKUSDW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO ((KL/2) - 1)

    i ← j \* 32

    IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

        THEN

            TMP\_SRC2[j+31:i] ← SRC2[31:0]

        ELSE

            TMP\_SRC2[j+31:i] ← SRC2[j+31:i]

    FI;

ENDFOR;

```

TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0];
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16];
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32];
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48];
TMP[79:64] ← (TMP_SRC2[31:0] < 0) ? 0 : TMP_SRC2[15:0];
DEST[79:64] ← (TMP_SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64];
TMP[95:80] ← (TMP_SRC2[63:32] < 0) ? 0 : TMP_SRC2[47:32];
DEST[95:80] ← (TMP_SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80];
TMP[111:96] ← (TMP_SRC2[95:64] < 0) ? 0 : TMP_SRC2[79:64];
DEST[111:96] ← (TMP_SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96];
TMP[127:112] ← (TMP_SRC2[127:96] < 0) ? 0 : TMP_SRC2[111:96];
DEST[127:112] ← (TMP_SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
IF VL >= 256
    TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
    DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128];
    TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
    DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144];

```

```

TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160];
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176];
TMP[207:192] ← (TMP_SRC2[159:128] < 0) ? 0 : TMP_SRC2[143:128];
DEST[207:192] ← (TMP_SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192];
TMP[223:208] ← (TMP_SRC2[191:160] < 0) ? 0 : TMP_SRC2[175:160];
DEST[223:208] ← (TMP_SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208];
TMP[239:224] ← (TMP_SRC2[223:192] < 0) ? 0 : TMP_SRC2[207:192];
DEST[239:224] ← (TMP_SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224];
TMP[255:240] ← (TMP_SRC2[255:224] < 0) ? 0 : TMP_SRC2[239:224];
DEST[255:240] ← (TMP_SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240];

```

FI;

IF VL >= 512

```

TMP[271:256] ← (SRC1[287:256] < 0) ? 0 : SRC1[271:256];
DEST[271:256] ← (SRC1[287:256] > FFFFH) ? FFFFH : TMP[271:256];
TMP[287:272] ← (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
DEST[287:272] ← (SRC1[319:288] > FFFFH) ? FFFFH : TMP[287:272];
TMP[303:288] ← (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
DEST[303:288] ← (SRC1[351:320] > FFFFH) ? FFFFH : TMP[303:288];
TMP[319:304] ← (SRC1[383:352] < 0) ? 0 : SRC1[367:352];
DEST[319:304] ← (SRC1[383:352] > FFFFH) ? FFFFH : TMP[319:304];
TMP[335:320] ← (TMP_SRC2[287:256] < 0) ? 0 : TMP_SRC2[271:256];
DEST[335:304] ← (TMP_SRC2[287:256] > FFFFH) ? FFFFH : TMP[79:64];
TMP[351:336] ← (TMP_SRC2[319:288] < 0) ? 0 : TMP_SRC2[303:288];
DEST[351:336] ← (TMP_SRC2[319:288] > FFFFH) ? FFFFH : TMP[351:336];
TMP[367:352] ← (TMP_SRC2[351:320] < 0) ? 0 : TMP_SRC2[315:320];
DEST[367:352] ← (TMP_SRC2[351:320] > FFFFH) ? FFFFH : TMP[367:352];
TMP[383:368] ← (TMP_SRC2[383:352] < 0) ? 0 : TMP_SRC2[367:352];
DEST[383:368] ← (TMP_SRC2[383:352] > FFFFH) ? FFFFH : TMP[383:368];
TMP[399:384] ← (SRC1[415:384] < 0) ? 0 : SRC1[399:384];
DEST[399:384] ← (SRC1[415:384] > FFFFH) ? FFFFH : TMP[399:384];
TMP[415:400] ← (SRC1[447:416] < 0) ? 0 : SRC1[431:416];
DEST[415:400] ← (SRC1[447:416] > FFFFH) ? FFFFH : TMP[415:400];
TMP[431:416] ← (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
DEST[431:416] ← (SRC1[479:448] > FFFFH) ? FFFFH : TMP[431:416];
TMP[447:432] ← (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
DEST[447:432] ← (SRC1[511:480] > FFFFH) ? FFFFH : TMP[447:432];
TMP[463:448] ← (TMP_SRC2[415:384] < 0) ? 0 : TMP_SRC2[399:384];
DEST[463:448] ← (TMP_SRC2[415:384] > FFFFH) ? FFFFH : TMP[463:448];
TMP[475:464] ← (TMP_SRC2[447:416] < 0) ? 0 : TMP_SRC2[431:416];
DEST[475:464] ← (TMP_SRC2[447:416] > FFFFH) ? FFFFH : TMP[475:464];
TMP[491:476] ← (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
DEST[491:476] ← (TMP_SRC2[479:448] > FFFFH) ? FFFFH : TMP[491:476];
TMP[511:492] ← (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];
DEST[511:492] ← (TMP_SRC2[511:480] > FFFFH) ? FFFFH : TMP[511:492];

```

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN

DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

```

        THEN *DEST[j+15:i] remains unchanged*
        ELSE *zeroing-masking*           ; zeroing-masking
            DEST[j+15:i] ← 0
    FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPACKUSDW__m512i__mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i__mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i__mm512_maskz_packus_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i__mm256_mask_packus_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i__mm256_maskz_packus_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i__mm_mask_packus_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKUSDW__m128i__mm_maskz_packus_epi32(__mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i__mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW__m256i__mm256_packus_epi32(__m256i m1, __m256i m2);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.



## PACKUSWB—Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 67 /r <sup>1</sup> PACKUSWB <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Converts 4 signed word integers from <i>mm</i> and 4 signed word integers from <i>mm/m64</i> into 8 unsigned byte integers in <i>mm</i> using unsigned saturation.
66 0F 67 /r PACKUSWB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Converts 8 signed word integers from <i>xmm1</i> and 8 signed word integers from <i>xmm2/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Converts 8 signed word integers from <i>xmm2</i> and 8 signed word integers from <i>xmm3/m128</i> into 16 unsigned byte integers in <i>xmm1</i> using unsigned saturation.
VEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Converts 16 signed word integers from <i>ymm2</i> and 16 signed word integers from <i>ymm3/m256</i> into 32 unsigned byte integers in <i>ymm1</i> using unsigned saturation.
EVEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB <i>xmm1{k1}{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Converts signed word integers from <i>xmm2</i> and signed word integers from <i>xmm3/m128</i> into unsigned byte integers in <i>xmm1</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB <i>ymm1{k1}{z}</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Converts signed word integers from <i>ymm2</i> and signed word integers from <i>ymm3/m256</i> into unsigned byte integers in <i>ymm1</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG 67 /r VPACKUSWB <i>zmm1{k1}{z}</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Converts signed word integers from <i>zmm2</i> and signed word integers from <i>zmm3/m512</i> into unsigned byte integers in <i>zmm1</i> using unsigned saturation under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Converts 4, 8, 16 or 32 signed word integers from the destination operand (first operand) and 4, 8, 16 or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 4-6 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.



VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

## Operation

### PACKUSWB (with 64-bit operands)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToUnsignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToUnsignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToUnsignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToUnsignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToUnsignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToUnsignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToUnsignedByte SRC[63:48];
```

### PACKUSWB (Legacy SSE instruction)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC[127:112]);
```

### PACKUSWB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
```

DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);  
 DEST[MAXVL-1:128] ← 0;

#### **VPACKUSWB (VEX.256 encoded version)**

DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);  
 DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);  
 DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);  
 DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);  
 DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);  
 DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);  
 DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);  
 DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);  
 DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);  
 DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);  
 DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);  
 DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);  
 DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);  
 DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);  
 DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);  
 DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);  
 DEST[135:128] ← SaturateSignedWordToUnsignedByte (SRC1[143:128]);  
 DEST[143:136] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);  
 DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);  
 DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);  
 DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);  
 DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);  
 DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);  
 DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);  
 DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);  
 DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);  
 DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);  
 DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);  
 DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);  
 DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);  
 DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);  
 DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);

#### **VPACKUSWB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

TMP\_DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);  
 TMP\_DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);  
 TMP\_DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);  
 TMP\_DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);  
 TMP\_DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);  
 TMP\_DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);  
 TMP\_DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);  
 TMP\_DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);  
 TMP\_DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);  
 TMP\_DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);  
 TMP\_DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);  
 TMP\_DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);  
 TMP\_DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);  
 TMP\_DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);

```

TMP_DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128] ← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    TMP_DEST[143:136] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    TMP_DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToUnsignedByte (SRC1[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToUnsignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToUnsignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToUnsignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToUnsignedByte (SRC2[479:464]);

```

```

    TMP_DEST[503:496] ← SaturateSignedWordToUnsignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
            ELSE *zeroing-masking*              ; zeroing-masking
                DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPACKUSWB __m512i __mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB __m512i __mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m512i __mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB __m256i __mm256_mask_packus_epi16(__m256i s, __mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m256i __mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB __m128i __mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
VPACKUSWB __m128i __mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);

PACKUSWB:    __m64 __mm_packs_pu16(__m64 m1, __m64 m2)
(V)PACKUSWB: __m128i __mm_packus_epi16(__m128i m1, __m128i m2)
VPACKUSWB:   __m256i __mm256_packus_epi16(__m256i m1, __m256i m2);

```

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

**PADDB/PADDW/PADDD/PADDQ—Add Packed Integers**

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
NP OF FC /r <sup>1</sup> PADDB <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Add packed byte integers from <i>mm/m64</i> and <i>mm</i> .
NP OF FD /r <sup>1</sup> PADDW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Add packed word integers from <i>mm/m64</i> and <i>mm</i> .
NP OF FE /r <sup>1</sup> PADDD <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Add packed doubleword integers from <i>mm/m64</i> and <i>mm</i> .
NP OF D4 /r <sup>1</sup> PADDQ <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Add packed quadword integers from <i>mm/m64</i> and <i>mm</i> .
66 OF FC /r PADDB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Add packed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> .
66 OF FD /r PADDW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Add packed word integers from <i>xmm2/m128</i> and <i>xmm1</i> .
66 OF FE /r PADDD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Add packed doubleword integers from <i>xmm2/m128</i> and <i>xmm1</i> .
66 OF D4 /r PADDQ <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Add packed quadword integers from <i>xmm2/m128</i> and <i>xmm1</i> .
VEEX.NDS.128.66.OF.WIG FC /r VPADDB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Add packed byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store in <i>xmm1</i> .
VEEX.NDS.128.66.OF.WIG FD /r VPADDW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Add packed word integers from <i>xmm2</i> , <i>xmm3/m128</i> and store in <i>xmm1</i> .
VEEX.NDS.128.66.OF.WIG FE /r VPADDD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Add packed doubleword integers from <i>xmm2</i> , <i>xmm3/m128</i> and store in <i>xmm1</i> .
VEEX.NDS.128.66.OF.WIG D4 /r VPADDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Add packed quadword integers from <i>xmm2</i> , <i>xmm3/m128</i> and store in <i>xmm1</i> .
VEEX.NDS.256.66.OF.WIG FC /r VPADDB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Add packed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store in <i>ymm1</i> .
VEEX.NDS.256.66.OF.WIG FD /r VPADDW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Add packed word integers from <i>ymm2</i> , <i>ymm3/m256</i> and store in <i>ymm1</i> .
VEEX.NDS.256.66.OF.WIG FE /r VPADDD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Add packed doubleword integers from <i>ymm2</i> , <i>ymm3/m256</i> and store in <i>ymm1</i> .
VEEX.NDS.256.66.OF.WIG D4 /r VPADDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Add packed quadword integers from <i>ymm2</i> , <i>ymm3/m256</i> and store in <i>ymm1</i> .
EVEX.NDS.128.66.OF.WIG FC /r VPADDB <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Add packed byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.128.66.OF.WIG FD /r VPADDW <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Add packed word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.128.66.OF.WO FE /r VPADDD <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	D	V/V	AVX512VL AVX512F	Add packed doubleword integers from <i>xmm2</i> , and <i>xmm3/m128/m32bcst</i> and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.128.66.OF.W1 D4 /r VPADDQ <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128/m64bcst</i>	D	V/V	AVX512VL AVX512F	Add packed quadword integers from <i>xmm2</i> , and <i>xmm3/m128/m64bcst</i> and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.OF.WIG FC /r VPADDB <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Add packed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.OF.WIG FD /r VPADDW <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Add packed word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store in <i>ymm1</i> using writemask <i>k1</i> .

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F.W0 FE /r VPADDD <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3</i> /m256/m32bcst	D	V/V	AVX512VL AVX512F	Add packed doubleword integers from <i>ymm2</i> , <i>ymm3</i> /m256/m32bcst and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W1 D4 /r VPADDQ <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3</i> /m256/m64bcst	D	V/V	AVX512VL AVX512F	Add packed quadword integers from <i>ymm2</i> , <i>ymm3</i> /m256/m64bcst and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG FC /r VPADDB <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3</i> /m512	C	V/V	AVX512BW	Add packed byte integers from <i>zmm2</i> , and <i>zmm3</i> /m512 and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG FD /r VPADDW <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3</i> /m512	C	V/V	AVX512BW	Add packed word integers from <i>zmm2</i> , and <i>zmm3</i> /m512 and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W0 FE /r VPADDD <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3</i> /m512/m32bcst	D	V/V	AVX512F	Add packed doubleword integers from <i>zmm2</i> , <i>zmm3</i> /m512/m32bcst and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W1 D4 /r VPADDQ <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3</i> /m512/m64bcst	D	V/V	AVX512F	Add packed quadword integers from <i>zmm2</i> , <i>zmm3</i> /m512/m64bcst and store in <i>zmm1</i> using writemask <i>k1</i> .
<b>NOTES:</b> 1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A</i> and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A</i> .				

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
B	NA	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
C	Full Mem	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
D	Full	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

### Description

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too



large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

**EVEX encoded VPADDD/Q:** The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

**EVEX encoded VPADDB/W:** The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

**VEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. the upper bits (MAXVL-1:256) of the destination are cleared.

**VEX.128 encoded version:** The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

**128-bit Legacy SSE version:** The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

## Operation

### PADDB (with 64-bit operands)

$DEST[7:0] \leftarrow DEST[7:0] + SRC[7:0];$

(\* Repeat add operation for 2nd through 7th byte \*)

$DEST[63:56] \leftarrow DEST[63:56] + SRC[63:56];$

### PADDW (with 64-bit operands)

$DEST[15:0] \leftarrow DEST[15:0] + SRC[15:0];$

(\* Repeat add operation for 2nd and 3th word \*)

$DEST[63:48] \leftarrow DEST[63:48] + SRC[63:48];$

### PADDD (with 64-bit operands)

$DEST[31:0] \leftarrow DEST[31:0] + SRC[31:0];$

$DEST[63:32] \leftarrow DEST[63:32] + SRC[63:32];$

### PADDQ (with 64-Bit operands)

$DEST[63:0] \leftarrow DEST[63:0] + SRC[63:0];$

### PADDB (Legacy SSE instruction)

$DEST[7:0] \leftarrow DEST[7:0] + SRC[7:0];$

(\* Repeat add operation for 2nd through 15th byte \*)

$DEST[127:120] \leftarrow DEST[127:120] + SRC[127:120];$

$DEST[MAXVL-1:128]$  (Unmodified)

### PADDW (Legacy SSE instruction)

$DEST[15:0] \leftarrow DEST[15:0] + SRC[15:0];$

(\* Repeat add operation for 2nd through 7th word \*)

$DEST[127:112] \leftarrow DEST[127:112] + SRC[127:112];$

$DEST[MAXVL-1:128]$  (Unmodified)

**PADD (Legacy SSE instruction)**

$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0] + \text{SRC}[31:0];$   
 (\* Repeat add operation for 2nd and 3th doubleword \*)  
 $\text{DEST}[127:96] \leftarrow \text{DEST}[127:96] + \text{SRC}[127:96];$   
 $\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

**PADDQ (Legacy SSE instruction)**

$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] + \text{SRC}[63:0];$   
 $\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] + \text{SRC}[127:64];$   
 $\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

**VPADDB (VEX.128 encoded instruction)**

$\text{DEST}[7:0] \leftarrow \text{SRC1}[7:0] + \text{SRC2}[7:0];$   
 (\* Repeat add operation for 2nd through 15th byte \*)  
 $\text{DEST}[127:120] \leftarrow \text{SRC1}[127:120] + \text{SRC2}[127:120];$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0;$

**VPADDW (VEX.128 encoded instruction)**

$\text{DEST}[15:0] \leftarrow \text{SRC1}[15:0] + \text{SRC2}[15:0];$   
 (\* Repeat add operation for 2nd through 7th word \*)  
 $\text{DEST}[127:112] \leftarrow \text{SRC1}[127:112] + \text{SRC2}[127:112];$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0;$

**VPADD (VEX.128 encoded instruction)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0];$   
 (\* Repeat add operation for 2nd and 3th doubleword \*)  
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] + \text{SRC2}[127:96];$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0;$

**VPADDQ (VEX.128 encoded instruction)**

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0];$   
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64];$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0;$

**VPADDB (VEX.256 encoded instruction)**

$\text{DEST}[7:0] \leftarrow \text{SRC1}[7:0] + \text{SRC2}[7:0];$   
 (\* Repeat add operation for 2nd through 31th byte \*)  
 $\text{DEST}[255:248] \leftarrow \text{SRC1}[255:248] + \text{SRC2}[255:248];$

**VPADDW (VEX.256 encoded instruction)**

$\text{DEST}[15:0] \leftarrow \text{SRC1}[15:0] + \text{SRC2}[15:0];$   
 (\* Repeat add operation for 2nd through 15th word \*)  
 $\text{DEST}[255:240] \leftarrow \text{SRC1}[255:240] + \text{SRC2}[255:240];$

**VPADD (VEX.256 encoded instruction)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] + \text{SRC2}[31:0];$   
 (\* Repeat add operation for 2nd and 7th doubleword \*)  
 $\text{DEST}[255:224] \leftarrow \text{SRC1}[255:224] + \text{SRC2}[255:224];$

**VPADDQ (VEX.256 encoded instruction)**

$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] + \text{SRC2}[63:0];$   
 $\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] + \text{SRC2}[127:64];$   
 $\text{DEST}[191:128] \leftarrow \text{SRC1}[191:128] + \text{SRC2}[191:128];$   
 $\text{DEST}[255:192] \leftarrow \text{SRC1}[255:192] + \text{SRC2}[255:192];$



**VPADDB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC1[i+7:i] + SRC2[i+7:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+7:i] = 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**VPADDW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC1[i+15:i] + SRC2[i+15:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] = 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**VPADDD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**VPADDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[63:0]

ELSE DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPADDB \_\_m512i \_mm512\_add\_epi8 (\_\_m512i a, \_\_m512i b)

VPADDW \_\_m512i \_mm512\_add\_epi16 (\_\_m512i a, \_\_m512i b)

VPADDB \_\_m512i \_mm512\_mask\_add\_epi8 (\_\_m512i s, \_\_mmask64 m, \_\_m512i a, \_\_m512i b)

VPADDW \_\_m512i \_mm512\_mask\_add\_epi16 (\_\_m512i s, \_\_mmask32 m, \_\_m512i a, \_\_m512i b)

VPADDB \_\_m512i \_mm512\_maskz\_add\_epi8 (\_\_mmask64 m, \_\_m512i a, \_\_m512i b)

VPADDW \_\_m512i \_mm512\_maskz\_add\_epi16 (\_\_mmask32 m, \_\_m512i a, \_\_m512i b)

VPADDB \_\_m256i \_mm256\_mask\_add\_epi8 (\_\_m256i s, \_\_mmask32 m, \_\_m256i a, \_\_m256i b)

VPADDW \_\_m256i \_mm256\_mask\_add\_epi16 (\_\_m256i s, \_\_mmask16 m, \_\_m256i a, \_\_m256i b)

VPADDB \_\_m256i \_mm256\_maskz\_add\_epi8 (\_\_mmask32 m, \_\_m256i a, \_\_m256i b)

VPADDW \_\_m256i \_mm256\_maskz\_add\_epi16 (\_\_mmask16 m, \_\_m256i a, \_\_m256i b)

VPADDB \_\_m128i \_mm\_mask\_add\_epi8 (\_\_m128i s, \_\_mmask16 m, \_\_m128i a, \_\_m128i b)

VPADDW \_\_m128i \_mm\_mask\_add\_epi16 (\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b)

VPADDB \_\_m128i \_mm\_maskz\_add\_epi8 (\_\_mmask16 m, \_\_m128i a, \_\_m128i b)

VPADDW \_\_m128i \_mm\_maskz\_add\_epi16 (\_\_mmask8 m, \_\_m128i a, \_\_m128i b)

VPADDD \_\_m512i \_mm512\_add\_epi32 (\_\_m512i a, \_\_m512i b);

VPADDD \_\_m512i \_mm512\_mask\_add\_epi32 (\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPADDD \_\_m512i \_mm512\_maskz\_add\_epi32 (\_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPADDD \_\_m256i \_mm256\_mask\_add\_epi32 (\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDD \_\_m256i \_mm256\_maskz\_add\_epi32 (\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDD \_\_m128i \_mm\_mask\_add\_epi32 (\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPADDD \_\_m128i \_mm\_maskz\_add\_epi32 (\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPADDQ \_\_m512i \_mm512\_add\_epi64 (\_\_m512i a, \_\_m512i b);

VPADDQ \_\_m512i \_mm512\_mask\_add\_epi64 (\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPADDQ \_\_m512i \_mm512\_maskz\_add\_epi64 (\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPADDQ \_\_m256i \_mm256\_mask\_add\_epi64 (\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDQ \_\_m256i \_mm256\_maskz\_add\_epi64 (\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPADDQ \_\_m128i \_mm\_mask\_add\_epi64 (\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPADDQ \_\_m128i \_mm\_maskz\_add\_epi64 (\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PADDB \_\_m128i \_mm\_add\_epi8 (\_\_m128i a, \_\_m128i b);

PADDW \_\_m128i \_mm\_add\_epi16 (\_\_m128i a, \_\_m128i b);

PADDD \_\_m128i \_mm\_add\_epi32 (\_\_m128i a, \_\_m128i b);

PADDDQ \_\_m128i \_mm\_add\_epi64 (\_\_m128i a, \_\_m128i b);

VPADDB \_\_m256i \_\_mm256\_add\_epi8 (\_\_m256ia, \_\_m256i b);  
 VPADDW \_\_m256i \_\_mm256\_add\_epi16 (\_\_m256i a, \_\_m256i b);  
 VPADDQ \_\_m256i \_\_mm256\_add\_epi32 (\_\_m256i a, \_\_m256i b);  
 VPADDQ \_\_m256i \_\_mm256\_add\_epi64 (\_\_m256i a, \_\_m256i b);  
 PADDB \_\_m64 \_\_mm\_add\_pi8(\_\_m64 m1, \_\_m64 m2)  
 PADDW \_\_m64 \_\_mm\_add\_pi16(\_\_m64 m1, \_\_m64 m2)  
 PADDQ \_\_m64 \_\_mm\_add\_pi32(\_\_m64 m1, \_\_m64 m2)  
 PADDQ \_\_m64 \_\_mm\_add\_pi64(\_\_m64 m1, \_\_m64 m2)

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPADDQ/Q, see Exceptions Type E4.

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb.

## PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF EC /r <sup>1</sup> PADDSB <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Add packed signed byte integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 OF EC /r PADDSB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Add packed signed byte integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.
NP OF ED /r <sup>1</sup> PADDSW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Add packed signed word integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 OF ED /r PADDSW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Add packed signed word integers from <i>xmm2/m128</i> and <i>xmm1</i> and saturate the results.
VEX.NDS.128.66.OF.WIG EC /r VPADDSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Add packed signed byte integers from <i>xmm3/m128</i> and <i>xmm2</i> and saturate the results.
VEX.NDS.128.66.OF.WIG ED /r VPADDSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Add packed signed word integers from <i>xmm3/m128</i> and <i>xmm2</i> and saturate the results.
VEX.NDS.256.66.OF.WIG EC /r VPADDSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Add packed signed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .
VEX.NDS.256.66.OF.WIG ED /r VPADDSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Add packed signed word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .
EVEX.NDS.128.66.OF.WIG EC /r VPADDSB <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Add packed signed byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.OF.WIG EC /r VPADDSB <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Add packed signed byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.OF.WIG EC /r VPADDSB <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Add packed signed byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> .
EVEX.NDS.128.66.OF.WIG ED /r VPADDSW <i>xmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Add packed signed word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.OF.WIG ED /r VPADDSW <i>ymm1</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Add packed signed word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.OF.WIG ED /r VPADDSW <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Add packed signed word integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

(V)PADD SB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADD SW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

## Operation

**PADD SB (with 64-bit operands)**

```
DEST[7:0] ← SaturateToSignedByte(DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte(DEST[63:56] + SRC[63:56]);
```

**PADD SB (with 128-bit operands)**

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
```

**VPADD SB (VEX.128 encoded version)**

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] ← 0
```

**VPADD SB (VEX.256 encoded version)**

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

**VPADDSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**PADDSW (with 64-bit operands)**

```

DEST[15:0] ← SaturateToSignedWord(DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord(DEST[63:48] + SRC[63:48]);

```

**PADDSW (with 128-bit operands)**

```

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] + SRC[127:112]);

```

**VPADDSW (VEX.128 encoded version)**

```

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[MAXVL-1:128] ← 0

```

**VPADDSW (VEX.256 encoded version)**

```

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])

```

**VPADDSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

PADDSB:     \_\_m64 \_mm\_adds\_pi8(\_\_m64 m1, \_\_m64 m2)  
 (V)PADDSB:  \_\_m128i \_mm\_adds\_epi8 (\_\_m128i a, \_\_m128i b)  
 VPADDSB:    \_\_m256i \_mm256\_adds\_epi8 (\_\_m256i a, \_\_m256i b)  
 PADDSW:     \_\_m64 \_mm\_adds\_pi16(\_\_m64 m1, \_\_m64 m2)  
 (V)PADDSW:  \_\_m128i \_mm\_adds\_epi16 (\_\_m128i a, \_\_m128i b)  
 VPADDSW:    \_\_m256i \_mm256\_adds\_epi16 (\_\_m256i a, \_\_m256i b)  
 VPADDSB\_\_m512i \_mm512\_adds\_epi8 (\_\_m512i a, \_\_m512i b)  
 VPADDSW\_\_m512i \_mm512\_adds\_epi16 (\_\_m512i a, \_\_m512i b)  
 VPADDSB\_\_m512i \_mm512\_mask\_adds\_epi8 (\_\_m512i s, \_\_mmask64 m, \_\_m512i a, \_\_m512i b)  
 VPADDSW\_\_m512i \_mm512\_mask\_adds\_epi16 (\_\_m512i s, \_\_mmask32 m, \_\_m512i a, \_\_m512i b)  
 VPADDSB\_\_m512i \_mm512\_maskz\_adds\_epi8 (\_\_mmask64 m, \_\_m512i a, \_\_m512i b)  
 VPADDSW\_\_m512i \_mm512\_maskz\_adds\_epi16 (\_\_mmask32 m, \_\_m512i a, \_\_m512i b)  
 VPADDSB\_\_m256i \_mm256\_mask\_adds\_epi8 (\_\_m256i s, \_\_mmask32 m, \_\_m256i a, \_\_m256i b)  
 VPADDSW\_\_m256i \_mm256\_mask\_adds\_epi16 (\_\_m256i s, \_\_mmask16 m, \_\_m256i a, \_\_m256i b)  
 VPADDSB\_\_m256i \_mm256\_maskz\_adds\_epi8 (\_\_mmask32 m, \_\_m256i a, \_\_m256i b)  
 VPADDSW\_\_m256i \_mm256\_maskz\_adds\_epi16 (\_\_mmask16 m, \_\_m256i a, \_\_m256i b)  
 VPADDSB\_\_m128i \_mm\_mask\_adds\_epi8 (\_\_m128i s, \_\_mmask16 m, \_\_m128i a, \_\_m128i b)  
 VPADDSW\_\_m128i \_mm\_mask\_adds\_epi16 (\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b)  
 VPADDSB\_\_m128i \_mm\_maskz\_adds\_epi8 (\_\_mmask16 m, \_\_m128i a, \_\_m128i b)  
 VPADDSW\_\_m128i \_mm\_maskz\_adds\_epi16 (\_\_mmask8 m, \_\_m128i a, \_\_m128i b)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF DC /r <sup>1</sup> PADDUSB <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Add packed unsigned byte integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 OF DC /r PADDUSB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Add packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> saturate the results.
NP OF DD /r <sup>1</sup> PADDUSW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Add packed unsigned word integers from <i>mm/m64</i> and <i>mm</i> and saturate the results.
66 OF DD /r PADDUSW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Add packed unsigned word integers from <i>xmm2/m128</i> to <i>xmm1</i> and saturate the results.
VEX.NDS.128.66.0F.WIG DC /r VPADDUSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Add packed unsigned byte integers from <i>xmm3/m128</i> to <i>xmm2</i> and saturate the results.
VEX.NDS.128.66.0F.WIG DD /r VPADDUSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Add packed unsigned word integers from <i>xmm3/m128</i> to <i>xmm2</i> and saturate the results.
VEX.NDS.256.66.0F.WIG DC /r VPADDUSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Add packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .
VEX.NDS.256.66.0F.WIG DD /r VPADDUSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Add packed unsigned word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG DC /r VPADDUSB <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG DC /r VPADDUSB <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG DC /r VPADDUSB <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Add packed unsigned byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> .
EVEX.NDS.128.66.0F.WIG DD /r VPADDUSW <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from <i>xmm2</i> , and <i>xmm3/m128</i> and store the saturated results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG DD /r VPADDUSW <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from <i>ymm2</i> , and <i>ymm3/m256</i> and store the saturated results in <i>ymm1</i> under writemask <i>k1</i> .



EVEX.NDS.512.66.0F.WIG DD /r VPADDUSW <i>zmm1</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Add packed unsigned word integers from <i>zmm2</i> , and <i>zmm3/m512</i> and store the saturated results in <i>zmm1</i> under writemask <i>k1</i> .
--	---	-----	----------	--

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation****PADDUSB (with 64-bit operands)**

```
DEST[7:0] ← SaturateToUnsignedByte(DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte(DEST[63:56] + SRC[63:56])
```

**PADDUSB (with 128-bit operands)**

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] + SRC[127:120]);
```

**VPADDUSB (VEX.128 encoded version)**

DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);  
 (\* Repeat subtract operation for 2nd through 14th bytes \*)  
 DEST[127:120] ← SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);  
 DEST[MAXVL-1:128] ← 0

**VPADDUSB (VEX.256 encoded version)**

DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);  
 (\* Repeat add operation for 2nd through 31st bytes \*)  
 DEST[255:248] ← SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);

**PADDUSW (with 64-bit operands)**

DEST[15:0] ← SaturateToUnsignedWord(DEST[15:0] + SRC[15:0] );  
 (\* Repeat add operation for 2nd and 3rd words \*)  
 DEST[63:48] ← SaturateToUnsignedWord(DEST[63:48] + SRC[63:48] );

**PADDUSW (with 128-bit operands)**

DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] + SRC[15:0]);  
 (\* Repeat add operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] + SRC[127:112]);

**VPADDUSW (VEX.128 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);  
 DEST[MAXVL-1:128] ← 0

**VPADDUSW (VEX.256 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);  
 (\* Repeat add operation for 2nd through 15th words \*)  
 DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])

**VPADDUSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] + SRC2[i+7:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+7:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+7:i] = 0

  FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**VPADDUSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] = 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

PADDUSB:    __m64 _mm_adds_pu8(__m64 m1, __m64 m2)
PADDUSW:    __m64 _mm_adds_pu16(__m64 m1, __m64 m2)
(V)PADDUSB: __m128i _mm_adds_epu8 (__m128i a, __m128i b)
(V)PADDUSW: __m128i _mm_adds_epu16 (__m128i a, __m128i b)
VPADDUSB:   __m256i _mm256_adds_epu8 (__m256i a, __m256i b)
VPADDUSW:   __m256i _mm256_adds_epu16 (__m256i a, __m256i b)
VPADDUSB__m512i __mm512_adds_epu8 (__m512i a, __m512i b)
VPADDUSW__m512i __mm512_adds_epu16 (__m512i a, __m512i b)
VPADDUSB__m512i __mm512_mask_adds_epu8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i __mm512_mask_adds_epu16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB__m512i __mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i __mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB__m256i __mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i __mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB__m256i __mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i __mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB__m128i __mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i __mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB__m128i __mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i __mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PALIGNR — Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 3A OF /r ib <sup>1</sup> PALIGNR <i>mm1</i> , <i>mm2/m64</i> , <i>imm8</i>	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in <i>imm8</i> into <i>mm1</i> .
66 OF 3A OF /r ib PALIGNR <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in <i>imm8</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF3A.WIG OF /r ib VPALIGNR <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	B	V/V	AVX	Concatenate <i>xmm2</i> and <i>xmm3/m128</i> , extract byte aligned result shifted to the right by constant value in <i>imm8</i> and result is stored in <i>xmm1</i> .
VEX.NDS.256.66.OF3A.WIG OF /r ib VPALIGNR <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	B	V/V	AVX2	Concatenate pairs of 16 bytes in <i>ymm2</i> and <i>ymm3/m256</i> into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in <i>imm8</i> from each intermediate result, and two 16-byte results are stored in <i>ymm1</i> .
EVEX.NDS.128.66.OF3A.WIG OF /r ib VPALIGNR <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	C	V/V	AVX512VL AVX512BW	Concatenate <i>xmm2</i> and <i>xmm3/m128</i> into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in <i>imm8</i> and result is stored in <i>xmm1</i> .
EVEX.NDS.256.66.OF3A.WIG OF /r ib VPALIGNR <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	C	V/V	AVX512VL AVX512BW	Concatenate pairs of 16 bytes in <i>ymm2</i> and <i>ymm3/m256</i> into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in <i>imm8</i> from each intermediate result, and two 16-byte results are stored in <i>ymm1</i> .
EVEX.NDS.512.66.OF3A.WIG OF /r ib VPALIGNR <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>zmm3/m512</i> , <i>imm8</i>	C	V/V	AVX512BW	Concatenate pairs of 16 bytes in <i>zmm2</i> and <i>zmm3/m512</i> into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in <i>imm8</i> from each intermediate result, and four 16-byte results are stored in <i>zmm1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX,

XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e. 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The `imm8[7:0]` is the common shift count

used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The `imm8[7:0]` is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

Note: VEX.L must be 0, otherwise the instruction will #UD.

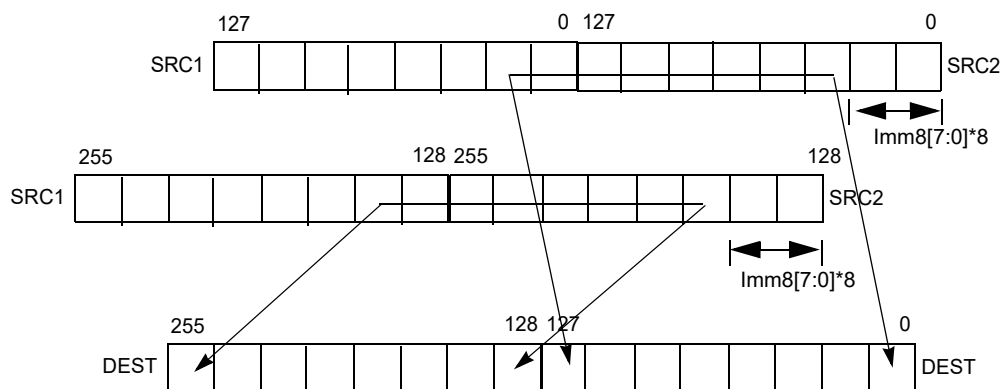


Figure 4-7. 256-bit VPALIGN Instruction Operation

## Operation

### PALIGNR (with 64-bit operands)

`temp1[127:0] = CONCATENATE(DEST, SRC) >> (imm8*8)`

`DEST[63:0] = temp1[63:0]`

**PALIGNR (with 128-bit operands)**

```
temp1[255:0] ← ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

**VPALIGNR (VEX.128 encoded version)**

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[MAXVL-1:128] ← 0
```

**VPALIGNR (VEX.256 encoded version)**

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] ← temp1[127:0]
temp1[255:0] ← ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[MAXVL-1:128] ← temp1[127:0]
```

**VPALIGNR (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR I ← 0 TO VL-1 with increments of 128

```
temp1[255:0] ← ((SRC1[I+127:I] << 128) OR SRC2[I+127:I])>>(imm8[7:0]*8);
TMP_DEST[I+127:I] ← temp1[127:0]
```

ENDFOR;

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TMP\_DEST[i+7:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] = 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

PALIGNR: `__m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)`

(V)PALIGNR: `__m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)`

VPALIGNR: `__m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)`

VPALIGNR `__m512i _mm512_alignr_epi8 (__m512i a, __m512i b, const int n)`

VPALIGNR `__m512i _mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)`

VPALIGNR `__m512i _mm512_maskz_alignr_epi8 (__mmask64 m, __m512i a, __m512i b, const int n)`

VPALIGNR `__m256i _mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)`

VPALIGNR `__m256i _mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)`

VPALIGNR `__m128i _mm_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)`

VPALIGNR `__m128i _mm_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)`

**SIMD Floating-Point Exceptions**

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF DB /r <sup>1</sup> PAND mm, mm/m64	A	V/V	MMX	Bitwise AND mm/m64 and mm.
66 OF DB /r PAND xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND of xmm2/m128 and xmm1.
VEX.NDS.128.66.OF.WIG DB /r VPAND xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND of xmm3/m128 and xmm.
VEX.NDS.256.66.OF.WIG DB /r VPAND ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise AND of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.NDS.128.66.OF.W0 DB /r VPANDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.OF.W0 DB /r VPANDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.OF.W0 DB /r VPANDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.128.66.OF.W1 DB /r VPANDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.OF.W1 DB /r VPANDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.OF.W1 DB /r VPANDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).



Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

## Operation

### PAND (64-bit operand)

DEST  $\leftarrow$  DEST AND SRC

### PAND (128-bit Legacy SSE version)

DEST  $\leftarrow$  DEST AND SRC

DEST[MAXVL-1:128] (Unmodified)

### VPAND (VEX.128 encoded version)

DEST  $\leftarrow$  SRC1 AND SRC2

DEST[MAXVL-1:128]  $\leftarrow$  0

### VPAND (VEX.256 encoded instruction)

DEST[255:0]  $\leftarrow$  (SRC1[255:0] AND SRC2[255:0])

DEST[MAXVL-1:256]  $\leftarrow$  0

### VPANDD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

    i  $\leftarrow$  j \* 32

    IF k1[j] OR \*no writemask\*

        THEN

            IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

                THEN DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE AND SRC2[31:0]

                ELSE DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE AND SRC2[i+31:i]

            FI;

        ELSE

            IF \*merging-masking\* ; merging-masking

                THEN \*DEST[i+31:i] remains unchanged\*

                ELSE ; zeroing-masking

                    DEST[i+31:i]  $\leftarrow$  0

            FI

    FI;

ENDFOR

DEST[MAXVL-1:VL]  $\leftarrow$  0

**VPANDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[63:0]
                ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPANDDD __m512i __mm512_and_epi32(__m512i a, __m512i b);
VPANDDD __m512i __mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDDD __m512i __mm512_maskz_and_epi32(__mmask16 k, __m512i a, __m512i b);
VPANDQ __m512i __mm512_and_epi64(__m512i a, __m512i b);
VPANDQ __m512i __mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDQ __m512i __mm512_maskz_and_epi64(__mmask8 k, __m512i a, __m512i b);
VPANDND __m256i __mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i __mm256_maskz_and_epi32(__mmask8 k, __m256i a, __m256i b);
VPANDND __m128i __mm_mask_and_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i __mm_maskz_and_epi32(__mmask8 k, __m128i a, __m128i b);
VPANDNQ __m256i __mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i __mm256_maskz_and_epi64(__mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i __mm_mask_and_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i __mm_maskz_and_epi64(__mmask8 k, __m128i a, __m128i b);
PAND: __m64 __mm_and_si64(__m64 m1, __m64 m2)
(V)PAND: __m128i __mm_and_si128(__m128i a, __m128i b)
VPAND: __m256i __mm256_and_si256(__m256i a, __m256i b)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF DF /r <sup>1</sup> PANDN mm, mm/m64	A	V/V	MMX	Bitwise AND NOT of mm/m64 and mm.
66 OF DF /r PANDN xmm1, xmm2/m128	A	V/V	SSE2	Bitwise AND NOT of xmm2/m128 and xmm1.
VEX.NDS.128.66.0F.WIG DF /r VPANDN xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND NOT of xmm3/m128 and xmm2.
VEX.NDS.256.66.0F.WIG DF /r VPANDN ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Bitwise AND NOT of ymm2, and ymm3/m256 and store result in ymm1.
EVEX.NDS.128.66.0F.W0 DF /r VPANDND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 DF /r VPANDND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 DF /r VPANDND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.W1 DF /r VPANDNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 DF /r VPANDNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 DF /r VPANDNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

## Operation

### PANDN (64-bit operand)

$DEST \leftarrow \text{NOT}(DEST) \text{ AND } SRC$

### PANDN (128-bit Legacy SSE version)

$DEST \leftarrow \text{NOT}(DEST) \text{ AND } SRC$

$DEST[\text{MAXVL}-1:128]$  (Unmodified)

### VPANDN (VEX.128 encoded version)

$DEST \leftarrow \text{NOT}(SRC1) \text{ AND } SRC2$

$DEST[\text{MAXVL}-1:128] \leftarrow 0$

### VPANDN (VEX.256 encoded instruction)

$DEST[255:0] \leftarrow ((\text{NOT } SRC1[255:0]) \text{ AND } SRC2[255:0])$

$DEST[\text{MAXVL}-1:256] \leftarrow 0$

### VPANDND (EVEX encoded versions)

$(KL, VL) = (4, 128), (8, 256), (16, 512)$

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

    IF  $k1[j]$  OR \*no writemask\*

        THEN

            IF  $(\text{EVEX}.b = 1) \text{ AND } (SRC2 \text{ *is memory*})$

                THEN  $DEST[i+31:i] \leftarrow ((\text{NOT } SRC1[i+31:i]) \text{ AND } SRC2[31:0])$

                ELSE  $DEST[i+31:i] \leftarrow ((\text{NOT } SRC1[i+31:i]) \text{ AND } SRC2[i+31:i])$

            FI;

        ELSE

            IF \*merging-masking\* ; merging-masking

                THEN \*DEST[i+31:i] remains unchanged\*

                ELSE ; zeroing-masking

$DEST[i+31:i] \leftarrow 0$

            FI

    FI;

ENDFOR

$DEST[\text{MAXVL}-1:VL] \leftarrow 0$

**VPANDNQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[63:0])

ELSE DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPANDND \_\_m512i \_mm512\_andnot\_epi32( \_\_m512i a, \_\_m512i b);

VPANDND \_\_m512i \_mm512\_mask\_andnot\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPANDND \_\_m512i \_mm512\_maskz\_andnot\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);

VPANDND \_\_m256i \_mm256\_mask\_andnot\_epi32( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDND \_\_m256i \_mm256\_maskz\_andnot\_epi32( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDND \_\_m128i \_mm\_mask\_andnot\_epi32( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDND \_\_m128i \_mm\_maskz\_andnot\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDNQ \_\_m512i \_mm512\_andnot\_epi64( \_\_m512i a, \_\_m512i b);

VPANDNQ \_\_m512i \_mm512\_mask\_andnot\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPANDNQ \_\_m512i \_mm512\_maskz\_andnot\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPANDNQ \_\_m256i \_mm256\_mask\_andnot\_epi64( \_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDNQ \_\_m256i \_mm256\_maskz\_andnot\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPANDNQ \_\_m128i \_mm\_mask\_andnot\_epi64( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPANDNQ \_\_m128i \_mm\_maskz\_andnot\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PANDN: \_\_m64 \_mm\_andnot\_si64( \_\_m64 m1, \_\_m64 m2)

(V)PANDN: \_\_m128i \_mm\_andnot\_si128( \_\_m128i a, \_\_m128i b)

VPANDN: \_\_m256i \_mm256\_andnot\_si256( \_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PAUSE—Spin Loop Hint

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F3 90	PAUSE	Z0	Valid	Valid	Gives hint to processor that improves performance of spin-wait loops.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Improves the performance of spin-wait loops. When executing a “spin-wait loop,” processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor’s power consumption.

This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

### Operation

Execute\_Next\_Instruction(Delay);

### Numeric Exceptions

None.

### Exceptions (All Operating Modes)

#UD If the LOCK prefix is used.

## PAVGB/PAVGW—Average Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF E0 /r <sup>1</sup> PAVGB <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSE	Average packed unsigned byte integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 OF E0, /r PAVGB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Average packed unsigned byte integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.
NP OF E3 /r <sup>1</sup> PAVGW <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSE	Average packed unsigned word integers from <i>mm2/m64</i> and <i>mm1</i> with rounding.
66 OF E3 /r PAVGW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Average packed unsigned word integers from <i>xmm2/m128</i> and <i>xmm1</i> with rounding.
VEX.NDS.128.66.OF.WIG E0 /r VPAVGB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Average packed unsigned byte integers from <i>xmm3/m128</i> and <i>xmm2</i> with rounding.
VEX.NDS.128.66.OF.WIG E3 /r VPAVGW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Average packed unsigned word integers from <i>xmm3/m128</i> and <i>xmm2</i> with rounding.
VEX.NDS.256.66.OF.WIG E0 /r VPAVGB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Average packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> with rounding and store to <i>ymm1</i> .
VEX.NDS.256.66.OF.WIG E3 /r VPAVGW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Average packed unsigned word integers from <i>ymm2</i> , <i>ymm3/m256</i> with rounding to <i>ymm1</i> .
EVEX.NDS.128.66.OF.WIG E0 /r VPAVGB <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from <i>xmm2</i> , and <i>xmm3/m128</i> with rounding and store to <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.OF.WIG E0 /r VPAVGB <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from <i>ymm2</i> , and <i>ymm3/m256</i> with rounding and store to <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.OF.WIG E0 /r VPAVGB <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Average packed unsigned byte integers from <i>zmm2</i> , and <i>zmm3/m512</i> with rounding and store to <i>zmm1</i> under writemask <i>k1</i> .
EVEX.NDS.128.66.OF.WIG E3 /r VPAVGW <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from <i>xmm2</i> , <i>xmm3/m128</i> with rounding to <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.OF.WIG E3 /r VPAVGW <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from <i>ymm2</i> , <i>ymm3/m256</i> with rounding to <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.OF.WIG E3 /r VPAVGW <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Average packed unsigned word integers from <i>zmm2</i> , <i>zmm3/m512</i> with rounding to <i>zmm1</i> under writemask <i>k1</i> .

## NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

## Operation

**PAVGB (with 64-bit operands)**

$DEST[7:0] \leftarrow (SRC[7:0] + DEST[7:0] + 1) \gg 1$ ; (\* Temp sum before shifting is 9 bits \*)  
 (\* Repeat operation performed for bytes 2 through 6 \*)  
 $DEST[63:56] \leftarrow (SRC[63:56] + DEST[63:56] + 1) \gg 1$ ;

**PAVGW (with 64-bit operands)**

$DEST[15:0] \leftarrow (SRC[15:0] + DEST[15:0] + 1) \gg 1$ ; (\* Temp sum before shifting is 17 bits \*)  
 (\* Repeat operation performed for words 2 and 3 \*)  
 $DEST[63:48] \leftarrow (SRC[63:48] + DEST[63:48] + 1) \gg 1$ ;

**PAVGB (with 128-bit operands)**

$DEST[7:0] \leftarrow (SRC[7:0] + DEST[7:0] + 1) \gg 1$ ; (\* Temp sum before shifting is 9 bits \*)  
 (\* Repeat operation performed for bytes 2 through 14 \*)  
 $DEST[127:120] \leftarrow (SRC[127:120] + DEST[127:120] + 1) \gg 1$ ;

**PAVGW (with 128-bit operands)**

$DEST[15:0] \leftarrow (SRC[15:0] + DEST[15:0] + 1) \gg 1$ ; (\* Temp sum before shifting is 17 bits \*)  
 (\* Repeat operation performed for words 2 through 6 \*)  
 $DEST[127:112] \leftarrow (SRC[127:112] + DEST[127:112] + 1) \gg 1$ ;



**VPAVGB (VEX.128 encoded version)**

```
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
(* Repeat operation performed for bytes 2 through 15 *)
DEST[127:120] ← (SRC1[127:120] + SRC2[127:120] + 1) >> 1
DEST[MAXVL-1:128] ← 0
```

**VPAVGW (VEX.128 encoded version)**

```
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
(* Repeat operation performed for 16-bit words 2 through 7 *)
DEST[127:112] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1
DEST[MAXVL-1:128] ← 0
```

**VPAVGB (VEX.256 encoded instruction)**

DEST[7:0]  $\leftarrow$  (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (\* Temp sum before shifting is 9 bits \*)  
 (\* Repeat operation performed for bytes 2 through 31)  
 DEST[255:248]  $\leftarrow$  (SRC1[255:248] + SRC2[255:248] + 1) >> 1;

**VPAVGW (VEX.256 encoded instruction)**

DEST[15:0]  $\leftarrow$  (SRC1[15:0] + SRC2[15:0] + 1)  $\gg$  1; (\* Temp sum before shifting is 17 bits \*)  
 (\* Repeat operation performed for words 2 through 15)  
 DEST[255:14]  $\leftarrow$  (SRC1[255:240] + SRC2[255:240] + 1)  $\gg$  1;

VPAVGB (EVEX encoded versions)

 $(KL, VL) = (16, 128), (32, 256), (64, 512)$ 

FOR  $j \leftarrow 0$  TO  $KL-1$

$$i \leftarrow i * 8$$

IF k1[j] OR \*no writemask\*

```
THEN DEST[j+7:i] ← (SRC1[j+7:i] + SRC2[j+7:i] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
```

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

```
ELSE *zeroing-masking* ; zeroing-masking
```

$$\text{DEST}[i+7:i] = 0$$

FI

Fl:

ENDFOR;

$$\text{DEST}[\text{MAXVL}-1:\text{VL}] \leftarrow 0$$

### VPAVGW (EVEX encoded versions)

 $(KL, VL) = (8, 128), (16, 256), (32, 512)$ 

FOR  $j \leftarrow 0$  TO  $KL-1$

$$i \leftarrow j * 16$$

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i]  $\leftarrow$  (SRC1[i+15:i] + SRC2[i+15:i] + 1)  $\gg$  1  
; (\* Temp sum before shifting is 17 bits \*)

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+15:i] remains unchanged\*

```
ELSE *zeroing-masking* ; zeroing-masking
```

$$\text{DEST}[i+15:i] = 0$$

FI

Fl:

ENDFOR;

$$\text{DEST}[\text{MAXVL}-1:\text{VL}] \leftarrow 0$$

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPAVGB __m512i _mm512_avg_epu8( __m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16( __m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__m512i s, __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__m512i s, __mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8( __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16( __mmask32 m, __m512i a, __m512i b);
VPAVGB __m256i _mm256_mask_avg_epu8(__m256i s, __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__m256i s, __mmask16 m, __m256i a, __m256i b);
VPAVGB __m256i _mm256_maskz_avg_epu8( __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16( __mmask16 m, __m256i a, __m256i b);
VPAVGB __m128i _mm_mask_avg_epu8(__m128i s, __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_mask_avg_epu16(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPAVGB __m128i _mm_maskz_avg_epu8( __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_maskz_avg_epu16( __mmask8 m, __m128i a, __m128i b);
PAVGB: __m64 _mm_avg_pu8 ( __m64 a, __m64 b)
PAVGW: __m64 _mm_avg_pu16 ( __m64 a, __m64 b)
(V)PAVGB: __m128i _mm_avg_epu8 ( __m128i a, __m128i b)
(V)PAVGW: __m128i _mm_avg_epu16 ( __m128i a, __m128i b)
VPAVGB:      __m256i _mm256_avg_epu8 ( __m256i a, __m256i b)
VPAVGW:      __m256i _mm256_avg_epu16 ( __m256i a, __m256i b)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PBLENDVB — Variable Blend Packed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB <i>xmm1</i> , <i>xmm2/m128</i> , < <i>XMM0</i> >	RM	V/V	SSE4_1	Select byte values from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in the high bit of each byte in <i>XMM0</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.W0 4C /r /is4 VPBLENDVB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>xmm4</i>	RVMR	V/V	AVX	Select byte values from <i>xmm2</i> and <i>xmm3/m128</i> using mask bits in the specified mask register, <i>xmm4</i> , and store the values into <i>xmm1</i> .
VEX.NDS.256.66.0F3A.W0 4C /r /is4 VPBLENDVB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>ymm4</i>	RVMR	V/V	AVX2	Select byte values from <i>ymm2</i> and <i>ymm3/m256</i> from mask specified in the high bit of each byte in <i>ymm4</i> and store the values into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	< <i>XMM0</i> >	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

### Description

Conditionally copies byte elements from the source operand (second operand) to the destination operand (first operand) depending on mask bits defined in the implicit third register argument, *XMM0*. The mask bits are the most significant bit in each byte element of the *XMM0* register.

If a mask bit is "1", then the corresponding byte element in the source operand is copied to the destination, else the byte element in the destination operand is left unchanged.

The register assignment of the implicit third operand is defined to be the architectural register *XMM0*.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register *XMM0*. An attempt to execute PBLENDVB with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and the destination operand are YMM registers. The second source operand is an YMM register or 256-bit memory location. The third source register is an YMM register and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats *XMM0* implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

### Operation

#### PBLENDVB (128-bit Legacy SSE version)

MASK ← *XMM0*

IF (MASK[7] = 1) THEN DEST[7:0] ← SRC[7:0];

ELSE DEST[7:0] ← DEST[7:0];

IF (MASK[15] = 1) THEN DEST[15:8] ← SRC[15:8];

```

ELSE DEST[15:8] ← DEST[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC[23:16]
ELSE DEST[23:16] ← DEST[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC[31:24]
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC[39:32]
ELSE DEST[39:32] ← DEST[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC[55:48]
ELSE DEST[55:48] ← DEST[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ← DEST[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ← DEST[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC[119:112]
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120])
DEST[MAXVL-1:128] (Unmodified)

```

#### **VPBLENDVB (VEX.128 encoded version)**

```

MASK ← SRC3
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC2[87:80]

```

```

ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
DEST[MAXVL-1:128] ← 0

```

**VPBLENDVB (VEX.256 encoded version)**

```

MASK ← SRC3
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC2[87:80]
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
IF (MASK[135] == 1) THEN DEST[135:128] ← SRC2[135:128];
ELSE DEST[135:128] ← SRC1[135:128];
IF (MASK[143] == 1) THEN DEST[143:136] ← SRC2[143:136];
ELSE DEST[143:136] ← SRC1[143:136];
IF (MASK[151] == 1) THEN DEST[151:144] ← SRC2[151:144]
ELSE DEST[151:144] ← SRC1[151:144];
IF (MASK[159] == 1) THEN DEST[159:152] ← SRC2[159:152]

```

```

ELSE DEST[159:152] ← SRC1[159:152];
IF (MASK[167] == 1) THEN DEST[167:160] ← SRC2[167:160]
ELSE DEST[167:160] ← SRC1[167:160];
IF (MASK[175] == 1) THEN DEST[175:168] ← SRC2[175:168]
ELSE DEST[175:168] ← SRC1[175:168];
IF (MASK[183] == 1) THEN DEST[183:176] ← SRC2[183:176]
ELSE DEST[183:176] ← SRC1[183:176];
IF (MASK[191] == 1) THEN DEST[191:184] ← SRC2[191:184]
ELSE DEST[191:184] ← SRC1[191:184];
IF (MASK[199] == 1) THEN DEST[199:192] ← SRC2[199:192]
ELSE DEST[199:192] ← SRC1[199:192];
IF (MASK[207] == 1) THEN DEST[207:200] ← SRC2[207:200]
ELSE DEST[207:200] ← SRC1[207:200];
IF (MASK[215] == 1) THEN DEST[215:208] ← SRC2[215:208]
ELSE DEST[215:208] ← SRC1[215:208];
IF (MASK[223] == 1) THEN DEST[223:216] ← SRC2[223:216]
ELSE DEST[223:216] ← SRC1[223:216];
IF (MASK[231] == 1) THEN DEST[231:224] ← SRC2[231:224]
ELSE DEST[231:224] ← SRC1[231:224];
IF (MASK[239] == 1) THEN DEST[239:232] ← SRC2[239:232]
ELSE DEST[239:232] ← SRC1[239:232];
IF (MASK[247] == 1) THEN DEST[247:240] ← SRC2[247:240]
ELSE DEST[247:240] ← SRC1[247:240];
IF (MASK[255] == 1) THEN DEST[255:248] ← SRC2[255:248]
ELSE DEST[255:248] ← SRC1[255:248]

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

(V)PBLENDVB:  __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);
VPBLENDVB:    __m256i _mm256_blendv_epi8 (__m256i v1, __m256i v2, __m256i mask);

```

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD                      If VEX.W = 1.

## PBLENDW — Blend Packed Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0E /r ib PBLENDW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Select words from <i>xmm1</i> and <i>xmm2/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.128.66.0F3A.WIG 0E /r ib VPBLENDW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	AVX	Select words from <i>xmm2</i> and <i>xmm3/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.256.66.0F3A.WIG 0E /r ib VPBLENDW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX2	Select words from <i>ymm2</i> and <i>ymm3/m256</i> from mask specified in <i>imm8</i> and store the values into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word element in the destination operand is unchanged.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### PBLENDW (128-bit Legacy SSE version)

```

IF (imm8[0] = 1) THEN DEST[15:0] ← SRC[15:0]
ELSE DEST[15:0] ← DEST[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] ← SRC[31:16]
ELSE DEST[31:16] ← DEST[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] ← SRC[47:32]
ELSE DEST[47:32] ← DEST[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] ← SRC[63:48]
ELSE DEST[63:48] ← DEST[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] ← SRC[79:64]
ELSE DEST[79:64] ← DEST[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] ← SRC[95:80]
ELSE DEST[95:80] ← DEST[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] ← SRC[111:96]
ELSE DEST[111:96] ← DEST[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] ← SRC[127:112]

```

ELSE DEST[127:112]  $\leftarrow$  DEST[127:112]

**VPBLENDW (VEX.128 encoded version)**

IF (imm8[0] = 1) THEN DEST[15:0]  $\leftarrow$  SRC2[15:0]  
 ELSE DEST[15:0]  $\leftarrow$  SRC1[15:0]  
 IF (imm8[1] = 1) THEN DEST[31:16]  $\leftarrow$  SRC2[31:16]  
 ELSE DEST[31:16]  $\leftarrow$  SRC1[31:16]  
 IF (imm8[2] = 1) THEN DEST[47:32]  $\leftarrow$  SRC2[47:32]  
 ELSE DEST[47:32]  $\leftarrow$  SRC1[47:32]  
 IF (imm8[3] = 1) THEN DEST[63:48]  $\leftarrow$  SRC2[63:48]  
 ELSE DEST[63:48]  $\leftarrow$  SRC1[63:48]  
 IF (imm8[4] = 1) THEN DEST[79:64]  $\leftarrow$  SRC2[79:64]  
 ELSE DEST[79:64]  $\leftarrow$  SRC1[79:64]  
 IF (imm8[5] = 1) THEN DEST[95:80]  $\leftarrow$  SRC2[95:80]  
 ELSE DEST[95:80]  $\leftarrow$  SRC1[95:80]  
 IF (imm8[6] = 1) THEN DEST[111:96]  $\leftarrow$  SRC2[111:96]  
 ELSE DEST[111:96]  $\leftarrow$  SRC1[111:96]  
 IF (imm8[7] = 1) THEN DEST[127:112]  $\leftarrow$  SRC2[127:112]  
 ELSE DEST[127:112]  $\leftarrow$  SRC1[127:112]  
 DEST[MAXVL-1:128]  $\leftarrow$  0

**VPBLENDW (VEX.256 encoded version)**

IF (imm8[0] == 1) THEN DEST[15:0]  $\leftarrow$  SRC2[15:0]  
 ELSE DEST[15:0]  $\leftarrow$  SRC1[15:0]  
 IF (imm8[1] == 1) THEN DEST[31:16]  $\leftarrow$  SRC2[31:16]  
 ELSE DEST[31:16]  $\leftarrow$  SRC1[31:16]  
 IF (imm8[2] == 1) THEN DEST[47:32]  $\leftarrow$  SRC2[47:32]  
 ELSE DEST[47:32]  $\leftarrow$  SRC1[47:32]  
 IF (imm8[3] == 1) THEN DEST[63:48]  $\leftarrow$  SRC2[63:48]  
 ELSE DEST[63:48]  $\leftarrow$  SRC1[63:48]  
 IF (imm8[4] == 1) THEN DEST[79:64]  $\leftarrow$  SRC2[79:64]  
 ELSE DEST[79:64]  $\leftarrow$  SRC1[79:64]  
 IF (imm8[5] == 1) THEN DEST[95:80]  $\leftarrow$  SRC2[95:80]  
 ELSE DEST[95:80]  $\leftarrow$  SRC1[95:80]  
 IF (imm8[6] == 1) THEN DEST[111:96]  $\leftarrow$  SRC2[111:96]  
 ELSE DEST[111:96]  $\leftarrow$  SRC1[111:96]  
 IF (imm8[7] == 1) THEN DEST[127:112]  $\leftarrow$  SRC2[127:112]  
 ELSE DEST[127:112]  $\leftarrow$  SRC1[127:112]  
 IF (imm8[0] == 1) THEN DEST[143:128]  $\leftarrow$  SRC2[143:128]  
 ELSE DEST[143:128]  $\leftarrow$  SRC1[143:128]  
 IF (imm8[1] == 1) THEN DEST[159:144]  $\leftarrow$  SRC2[159:144]  
 ELSE DEST[159:144]  $\leftarrow$  SRC1[159:144]  
 IF (imm8[2] == 1) THEN DEST[175:160]  $\leftarrow$  SRC2[175:160]  
 ELSE DEST[175:160]  $\leftarrow$  SRC1[175:160]  
 IF (imm8[3] == 1) THEN DEST[191:176]  $\leftarrow$  SRC2[191:176]  
 ELSE DEST[191:176]  $\leftarrow$  SRC1[191:176]  
 IF (imm8[4] == 1) THEN DEST[207:192]  $\leftarrow$  SRC2[207:192]  
 ELSE DEST[207:192]  $\leftarrow$  SRC1[207:192]  
 IF (imm8[5] == 1) THEN DEST[223:208]  $\leftarrow$  SRC2[223:208]  
 ELSE DEST[223:208]  $\leftarrow$  SRC1[223:208]  
 IF (imm8[6] == 1) THEN DEST[239:224]  $\leftarrow$  SRC2[239:224]  
 ELSE DEST[239:224]  $\leftarrow$  SRC1[239:224]  
 IF (imm8[7] == 1) THEN DEST[255:240]  $\leftarrow$  SRC2[255:240]  
 ELSE DEST[255:240]  $\leftarrow$  SRC1[255:240]



### Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDW: `__m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);`

VPBLENDW: `__m256i _mm256_blend_epi16 (__m256i v1, __m256i v2, const int mask)`

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1 and AVX2 = 0.

## PCLMULQDQ — Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	PCLMUL- QDQ	Carry-less multiplication of one quadword of <i>xmm1</i> by one quadword of <i>xmm2/m128</i> , stores the 128-bit result in <i>xmm1</i> . The immediate is used to determine which quadwords of <i>xmm1</i> and <i>xmm2/m128</i> should be used.
VEX.NDS.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i> , <i>imm8</i>	RVMI	V/V	Both PCL- MULQDQ and AVX flags	Carry-less multiplication of one quadword of <i>xmm2</i> by one quadword of <i>xmm3/m128</i> , stores the 128-bit result in <i>xmm1</i> . The immediate is used to determine which quadwords of <i>xmm2</i> and <i>xmm3/m128</i> should be used.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-13, other bits of the immediate byte are ignored.

**Table 4-13. PCLMULQDQ Quadword Selection of Immediate Byte**

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL( SRC2 <sup>1</sup> [63:0], SRC1[63:0] )
0	1	CL_MUL( SRC2[63:0], SRC1[127:64] )
1	0	CL_MUL( SRC2[127:64], SRC1[63:0] )
1	1	CL_MUL( SRC2[127:64], SRC1[127:64] )

### NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simplify programming and emit the required encoding for Imm8.

**Table 4-14. Pseudo-Op and PCLMULQDQ Implementation**

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ <i>xmm1</i> , <i>xmm2</i>	0000_0000B
PCLMULHQLQDQ <i>xmm1</i> , <i>xmm2</i>	0000_0001B
PCLMULLQHQQDQ <i>xmm1</i> , <i>xmm2</i>	0001_0000B
PCLMULHQHQQDQ <i>xmm1</i> , <i>xmm2</i>	0001_0001B

**Operation****PCLMULQDQ**

```

IF (Imm8[0] = 0 )
    THEN
        TEMP1  $\leftarrow$  SRC1 [63:0];
    ELSE
        TEMP1  $\leftarrow$  SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2  $\leftarrow$  SRC2 [63:0];
    ELSE
        TEMP2  $\leftarrow$  SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ]  $\leftarrow$  (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [ i ]  $\leftarrow$  TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ]  $\leftarrow$  TmpB[ i ];
}
For i = 64 to 126 {
    TmpB [ i ]  $\leftarrow$  0;
    For j = i - 63 to 63 {
        TmpB [ i ]  $\leftarrow$  TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ]  $\leftarrow$  TmpB[ i ];
}
DEST[127]  $\leftarrow$  0;
DEST[MAXVL-1:128] (Unmodified)

```

**VPCLMULQDQ**

```

IF (Imm8[0] = 0 )
    THEN
        TEMP1  $\leftarrow$  SRC1 [63:0];
    ELSE
        TEMP1  $\leftarrow$  SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2  $\leftarrow$  SRC2 [63:0];
    ELSE
        TEMP2  $\leftarrow$  SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ]  $\leftarrow$  (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [ i ]  $\leftarrow$  TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[i]  $\leftarrow$  TmpB[i];
}
For i = 64 to 126 {
    TmpB [ i ]  $\leftarrow$  0;
    For j = i - 63 to 63 {

```

```

    TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
  }
  DEST[i] ← TmpB[i];
}
DEST[MAXVL-1:127] ← 0;

```

### Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ: `__m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4, additionally

#UD                      If VEX.L = 1.

## PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 74 /r <sup>1</sup> PCMPEQB <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Compare packed bytes in <i>mm/m64</i> and <i>mm</i> for equality.
66 OF 74 /r PCMPEQB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Compare packed bytes in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
NP OF 75 /r <sup>1</sup> PCMPEQW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Compare packed words in <i>mm/m64</i> and <i>mm</i> for equality.
66 OF 75 /r PCMPEQW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Compare packed words in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
NP OF 76 /r <sup>1</sup> PCMPEQD <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Compare packed doublewords in <i>mm/m64</i> and <i>mm</i> for equality.
66 OF 76 /r PCMPEQD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Compare packed doublewords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
VEX.NDS.128.66.OF.WIG 74 /r VPCMPEQB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed bytes in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.128.66.OF.WIG 75 /r VPCMPEQW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed words in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.128.66.OF.WIG 76 /r VPCMPEQD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed doublewords in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.256.66.OF.WIG 74 /r VPCMPEQB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3 /m256</i>	B	V/V	AVX2	Compare packed bytes in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
VEX.NDS.256.66.OF.WIG 75 /r VPCMPEQW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3 /m256</i>	B	V/V	AVX2	Compare packed words in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
VEX.NDS.256.66.OF.WIG 76 /r VPCMPEQD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3 /m256</i>	B	V/V	AVX2	Compare packed doublewords in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
EVEX.NDS.128.66.OF.WO 76 /r VPCMPEQD <i>k1 {k2}</i> , <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	C	V/V	AVX512VL AVX512F	Compare Equal between int32 vector <i>xmm2</i> and int32 vector <i>xmm3/m128/m32bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.OF.WO 76 /r VPCMPEQD <i>k1 {k2}</i> , <i>ymm2</i> , <i>ymm3/m256/m32bcst</i>	C	V/V	AVX512VL AVX512F	Compare Equal between int32 vector <i>ymm2</i> and int32 vector <i>ymm3/m256/m32bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.OF.WO 76 /r VPCMPEQD <i>k1 {k2}</i> , <i>zmm2</i> , <i>zmm3/m512/m32bcst</i>	C	V/V	AVX512F	Compare Equal between int32 vectors in <i>zmm2</i> and <i>zmm3/m512/m32bcst</i> , and set destination <i>k1</i> according to the comparison results under writemask <i>k2</i> .
EVEX.NDS.128.66.OF.WIG 74 /r VPCMPEQB <i>k1 {k2}</i> , <i>xmm2</i> , <i>xmm3 /m128</i>	D	V/V	AVX512VL AVX512BW	Compare packed bytes in <i>xmm3/m128</i> and <i>xmm2</i> for equality and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, ymm2, ymm3 /m256	D	V/V	AVX512VL AVX512BW	Compare packed bytes in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW	Compare packed bytes in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, xmm2, xmm3 /m128	D	V/V	AVX512VL AVX512BW	Compare packed words in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, ymm2, ymm3 /m256	D	V/V	AVX512VL AVX512BW	Compare packed words in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW	Compare packed words in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVE encoded VPCMPEQD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVE encoded VPCMPEQB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

## Operation

### PCMPEQB (with 64-bit operands)

```
IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
IF DEST[63:56] = SRC[63:56]
    THEN DEST[63:56] ← FFH;
    ELSE DEST[63:56] ← 0; FI;
```

### COMPARE\_BYTES\_EQUAL (SRC1, SRC2)

```
IF SRC1[7:0] = SRC2[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
IF SRC1[127:120] = SRC2[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;
```

### COMPARE\_WORDS\_EQUAL (SRC1, SRC2)

```
IF SRC1[15:0] = SRC2[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
IF SRC1[127:112] = SRC2[127:112]
    THEN DEST[127:112] ← FFFFH;
    ELSE DEST[127:112] ← 0; FI;
```

### COMPARE\_DWORDS\_EQUAL (SRC1, SRC2)

```
IF SRC1[31:0] = SRC2[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
IF SRC1[127:96] = SRC2[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 0; FI;
```

### PCMPEQB (with 128-bit operands)

```
DEST[127:0] ← COMPARE_BYTES_EQUAL(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

**VPCMPEQB (VEX.128 encoded version)**

DEST[127:0] ← COMPARE\_BYTES\_EQUAL(SRC1[127:0], SRC2[127:0])

DEST[MAXVL-1:128] ← 0

**VPCMPEQB (VEX.256 encoded version)**

DEST[127:0] ← COMPARE\_BYTES\_EQUAL(SRC1[127:0], SRC2[127:0])

DEST[255:128] ← COMPARE\_BYTES\_EQUAL(SRC1[255:128], SRC2[255:128])

DEST[MAXVL-1:256] ← 0

**VPCMPEQB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k2[j] OR \*no writemask\*

    THEN

      /\* signed comparison \*/

      CMP ← SRC1[i+7:i] == SRC2[i+7:i];

      IF CMP = TRUE

        THEN DEST[j] ← 1;

        ELSE DEST[j] ← 0; FI;

      ELSE DEST[j] ← 0 ; zeroing-masking only FI;

    FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**PCMPEQW (with 64-bit operands)**

IF DEST[15:0] = SRC[15:0]

  THEN DEST[15:0] ← FFFFH;

  ELSE DEST[15:0] ← 0; FI;

(\* Continue comparison of 2nd and 3rd words in DEST and SRC \*)

IF DEST[63:48] = SRC[63:48]

  THEN DEST[63:48] ← FFFFH;

  ELSE DEST[63:48] ← 0; FI;

**PCMPEQW (with 128-bit operands)**

DEST[127:0] ← COMPARE\_WORDS\_EQUAL(DEST[127:0], SRC[127:0])

DEST[MAXVL-1:128] (Unmodified)

**VPCMPEQW (VEX.128 encoded version)**

DEST[127:0] ← COMPARE\_WORDS\_EQUAL(SRC1[127:0], SRC2[127:0])

DEST[MAXVL-1:128] ← 0

**VPCMPEQW (VEX.256 encoded version)**

DEST[127:0] ← COMPARE\_WORDS\_EQUAL(SRC1[127:0], SRC2[127:0])

DEST[255:128] ← COMPARE\_WORDS\_EQUAL(SRC1[255:128], SRC2[255:128])

DEST[MAXVL-1:256] ← 0



**VPCMPEQW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k2[j] OR \*no writemask\*

THEN

/\* signed comparison \*/

CMP ← SRC1[i+15:i] == SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only FI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**PCMPEQD (with 64-bit operands)**

IF DEST[31:0] = SRC[31:0]

THEN DEST[31:0] ← FFFFFFFFH;

ELSE DEST[31:0] ← 0; FI;

IF DEST[63:32] = SRC[63:32]

THEN DEST[63:32] ← FFFFFFFFH;

ELSE DEST[63:32] ← 0; FI;

**PCMPEQD (with 128-bit operands)**

DEST[127:0] ← COMPARE\_DWORDS\_EQUAL(DEST[127:0], SRC[127:0])

DEST[MAXVL-1:128] (Unmodified)

**VPCMPEQD (VEX.128 encoded version)**

DEST[127:0] ← COMPARE\_DWORDS\_EQUAL(SRC1[127:0], SRC2[127:0])

DEST[MAXVL-1:128] ← 0

**VPCMPEQD (VEX.256 encoded version)**

DEST[127:0] ← COMPARE\_DWORDS\_EQUAL(SRC1[127:0], SRC2[127:0])

DEST[255:128] ← COMPARE\_DWORDS\_EQUAL(SRC1[255:128], SRC2[255:128])

DEST[MAXVL-1:256] ← 0

**VPCMPEQD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k2[j] OR \*no writemask\*

THEN

/\* signed comparison \*/

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+31:i] = SRC2[31:0];

ELSE CMP ← SRC1[i+31:i] = SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

## Intel C/C++ Compiler Intrinsic Equivalents

```

VPCMPEQB __mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b);
VPCMPEQB __mmask64 _mm512_mask_cmpeq_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPEQB __mmask32 _mm256_cmpeq_epi8_mask(__m256i a, __m256i b);
VPCMPEQB __mmask32 _mm256_mask_cmpeq_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPEQB __mmask16 _mm_cmpeq_epi8_mask(__m128i a, __m128i b);
VPCMPEQB __mmask16 _mm_mask_cmpeq_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPEQW __mmask32 _mm512_cmpeq_epi16_mask(__m512i a, __m512i b);
VPCMPEQW __mmask32 _mm512_mask_cmpeq_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPEQW __mmask16 _mm256_cmpeq_epi16_mask(__m256i a, __m256i b);
VPCMPEQW __mmask16 _mm256_mask_cmpeq_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPEQW __mmask8 _mm_cmpeq_epi16_mask(__m128i a, __m128i b);
VPCMPEQW __mmask8 _mm_mask_cmpeq_epi16_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPEQD __mmask16 _mm512_cmpeq_epi32_mask(__m512i a, __m512i b);
VPCMPEQD __mmask16 _mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPEQD __mmask8 _mm256_cmpeq_epi32_mask(__m256i a, __m256i b);
VPCMPEQD __mmask8 _mm256_mask_cmpeq_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQD __mmask8 _mm_cmpeq_epi32_mask(__m128i a, __m128i b);
VPCMPEQD __mmask8 _mm_mask_cmpeq_epi32_mask(__mmask8 k, __m128i a, __m128i b);
PCMPEQB: __m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW: __m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD: __m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)
(V)PCMPEQB: __m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)
(V)PCMPEQW: __m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)
(V)PCMPEQD: __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)
VPCMPEQB:      __m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)
VPCMPEQW:      __m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)
VPCMPEQD:      __m256i _mm256_cmpeq_epi32 (__m256i a, __m256i b)

```

## Flags Affected

None.

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.  
 EVEX-encoded VPCMPEQD, see Exceptions Type E4.  
 EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb.

## PCMPEQQ — Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE4_1	Compare packed qwords in <i>xmm2/m128</i> and <i>xmm1</i> for equality.
VEX.NDS.128.66.0F38.WIG 29 /r VPCMPEQQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed quadwords in <i>xmm3/m128</i> and <i>xmm2</i> for equality.
VEX.NDS.256.66.0F38.WIG 29 /r VPCMPEQQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Compare packed quadwords in <i>ymm3/m256</i> and <i>ymm2</i> for equality.
EVEX.NDS.128.66.0F38.W1 29 /r VPCMPEQQ <i>k1 {k2}</i> , <i>xmm2</i> , <i>xmm3/m128/m64bcst</i>	C	V/V	AVX512VL AVX512F	Compare Equal between int64 vector <i>xmm2</i> and int64 vector <i>xmm3/m128/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F38.W1 29 /r VPCMPEQQ <i>k1 {k2}</i> , <i>ymm2</i> , <i>ymm3/m256/m64bcst</i>	C	V/V	AVX512VL AVX512F	Compare Equal between int64 vector <i>ymm2</i> and int64 vector <i>ymm3/m256/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F38.W1 29 /r VPCMPEQQ <i>k1 {k2}</i> , <i>zmm2</i> , <i>zmm3/m512/m64bcst</i>	C	V/V	AVX512F	Compare Equal between int64 vector <i>zmm2</i> and int64 vector <i>zmm3/m512/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQQ: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask *k2*.

## Operation

### PCMPEQQ (with 128-bit operands)

```

IF (DEST[63:0] = SRC[63:0])
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI;
IF (DEST[127:64] = SRC[127:64])
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI;
DEST[MAXVL-1:128] (Unmodified)

```

### COMPARE\_QWORDS\_EQUAL (SRC1, SRC2)

```

IF SRC1[63:0] = SRC2[63:0]
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI;
IF SRC1[127:64] = SRC2[127:64]
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI;

```

### VPCMPEQQ (VEX.128 encoded version)

```

DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

```

### VPCMPEQQ (VEX.256 encoded version)

```

DEST[127:0] ← COMPARE_QWORDS_EQUAL(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_QWORDS_EQUAL(SRC1[255:128], SRC2[255:128])
DEST[MAXVL-1:256] ← 0

```

### VPCMPEQQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+63:i] = SRC2[63:0];
                ELSE CMP ← SRC1[i+63:i] = SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
            ELSE    DEST[j] ← 0                ; zeroing-masking only
        FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPEQQ \_\_mmask8 \_mm512\_cmpeq\_epi64\_mask( \_\_m512i a, \_\_m512i b);  
 VPCMPEQQ \_\_mmask8 \_mm512\_mask\_cmpeq\_epi64\_mask(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPCMPEQQ \_\_mmask8 \_mm256\_cmpeq\_epi64\_mask( \_\_m256i a, \_\_m256i b);  
 VPCMPEQQ \_\_mmask8 \_mm256\_mask\_cmpeq\_epi64\_mask(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPCMPEQQ \_\_mmask8 \_mm\_cmpeq\_epi64\_mask( \_\_m128i a, \_\_m128i b);  
 VPCMPEQQ \_\_mmask8 \_mm\_mask\_cmpeq\_epi64\_mask(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 (V)PCMPEQQ: \_\_m128i \_mm\_cmpeq\_epi64(\_\_m128i a, \_\_m128i b);  
 VPCMPEQQ: \_\_m256i \_mm256\_cmpeq\_epi64( \_\_m256i a, \_\_m256i b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPEQQ, see Exceptions Type E4.

## PCMPESTRI — Packed Compare Explicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 61 /r imm8 PCMPESTRI <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A 61 /r ib VPCMPESTRI <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

### Description

The instruction compares and processes data from two string fragments based on the encoded value in the Imm8 Control Byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMP-ISTRM”), and generates an index stored to the count register (ECX).

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 (see Section 4.1.4) is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise  
 ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise  
 SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise  
 OFlag – IntRes2[0]  
 AFlag – Reset  
 PFlag – Reset

### Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Length 1	Length 2	Result
16 bit	xmm	xmm/m128	EAX	EDX	ECX
32 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	ECX

### Intel C/C++ Compiler Intrinsic Equivalent for Returning Index

```
int __mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);
```

**Intel C/C++ Compiler Intrinsics For Reading EFlag Results**

```

int  _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD                      If VEX.L = 1.  
                           If VEX.vvvv ≠ 1111B.

## PCMPESTRM — Packed Compare Explicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 60 /r imm8 PCMPESTRM <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in <i>XMM0</i> .
VEX.128.66.0F3A 60 /r ib VPCMPESTRM <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in <i>XMM0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

### Description

The instruction compares data from two string fragments based on the encoded value in the imm8 control byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM”), and generates a mask stored to *XMM0*.

Each string fragment is represented by two values. The first value is an *xmm* (or possibly *m128* for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is *EAX/RAX* (for *xmm1*) or *EDX/RDX* (for *xmm2/m128*). The length represents the number of bytes/words which are valid for the respective *xmm/m128* data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of *imm8[bit3]* when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by *imm8[6]*, *IntRes2* is then either stored to the least significant bits of *XMM0* (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to *XMM0*.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if *IntRes2* is equal to zero, set otherwise  
 ZFlag – Set if absolute-value of *EDX* is < 16 (8), reset otherwise  
 SFlag – Set if absolute-value of *EAX* is < 16 (8), reset otherwise  
 OFlag – *IntRes2[0]*  
 AFlag – Reset  
 PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of *XMM0* are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.



Effective Operand Size

Operating mode/size	Operand1	Operand 2	Length1	Length2	Result
16 bit	xmm	xmm/m128	EAX	EDX	XMM0
32 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

```
__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);
```

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

```
int  _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int  _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD                    If VEX.L = 1.  
                      If VEX.vvvv ≠ 1111B.

## PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 64 /r <sup>1</sup> PCMPGTB <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Compare packed signed byte integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 OF 64 /r PCMPGTB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Compare packed signed byte integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
NP OF 65 /r <sup>1</sup> PCMPGTW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Compare packed signed word integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 OF 65 /r PCMPGTW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Compare packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
NP OF 66 /r <sup>1</sup> PCMPGTD <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Compare packed signed doubleword integers in <i>mm</i> and <i>mm/m64</i> for greater than.
66 OF 66 /r PCMPGTD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Compare packed signed doubleword integers in <i>xmm1</i> and <i>xmm2/m128</i> for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMPGTD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed signed doubleword integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
VEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Compare packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
VEX.NDS.256.66.0F.WIG 66 /r VPCMPGTD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Compare packed signed doubleword integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
EVEX.NDS.128.66.0F.WO 66 /r VPCMPGTD <i>k1</i> { <i>k2</i> }, <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	C	V/V	AVX512VL AVX512F	Compare Greater between int32 vector <i>xmm2</i> and int32 vector <i>xmm3/m128/m32bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WO 66 /r VPCMPGTD <i>k1</i> { <i>k2</i> }, <i>ymm2</i> , <i>ymm3/m256/m32bcst</i>	C	V/V	AVX512VL AVX512F	Compare Greater between int32 vector <i>ymm2</i> and int32 vector <i>ymm3/m256/m32bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WO 66 /r VPCMPGTD <i>k1</i> { <i>k2</i> }, <i>zmm2</i> , <i>zmm3/m512/m32bcst</i>	C	V/V	AVX512F	Compare Greater between int32 elements in <i>zmm2</i> and <i>zmm3/m512/m32bcst</i> , and set destination <i>k1</i> according to the comparison results under writemask. <i>k2</i> .
EVEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB <i>k1</i> { <i>k2</i> }, <i>xmm2</i> , <i>xmm3/m128</i>	D	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in <i>xmm2</i> and <i>xmm3/m128</i> for greater than, and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB <i>k1</i> { <i>k2</i> }, <i>ymm2</i> , <i>ymm3/m256</i>	D	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in <i>ymm2</i> and <i>ymm3/m256</i> for greater than, and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.NDS.512.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, xmm2, xmm3/m128	D	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, ymm2, ymm3/m256	D	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding data element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

**EVEX encoded VPCMPGTD:** The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

**EVEX encoded VPCMPGTB/W:** The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

## Operation

### PCMPGTB (with 64-bit operands)

```
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
IF DEST[63:56] > SRC[63:56]
    THEN DEST[63:56] ← FFH;
    ELSE DEST[63:56] ← 0; FI;
```

### COMPARE\_BYTES\_GREATER (SRC1, SRC2)

```
IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;
```

### COMPARE\_WORDS\_GREATER (SRC1, SRC2)

```
IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] ← FFFFH;
    ELSE DEST[127:112] ← 0; FI;
```

### COMPARE\_DWORDS\_GREATER (SRC1, SRC2)

```
IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 0; FI;
```

### PCMPGTB (with 128-bit operands)

```
DEST[127:0] ← COMPARE_BYTES_GREATER(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

### VPCMPGTB (VEX.128 encoded version)

```
DEST[127:0] ← COMPARE_BYTES_GREATER(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0
```

**VPCMPGTB (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_BYTES_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_BYTES_GREATER(SRC1[255:128], SRC2[255:128])
DEST[MAXVL-1:256] ← 0

```

**VPCMPGTB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+7:i] > SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
            ELSE DEST[j] ← 0 ; zeroing-masking only FI;
        FI;
    ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**PCMPGTW (with 64-bit operands)**

```

IF DEST[15:0] > SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] > SRC[63:48]
    THEN DEST[63:48] ← FFFFH;
    ELSE DEST[63:48] ← 0; FI;

```

**PCMPGTW (with 128-bit operands)**

```

DEST[127:0] ← COMPARE_WORDS_GREATER(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

```

**VPCMPGTW (VEX.128 encoded version)**

```

DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

```

**VPCMPGTW (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_WORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_WORDS_GREATER(SRC1[255:128], SRC2[255:128])
DEST[MAXVL-1:256] ← 0

```

**VPCMPGTW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+15:i] > SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        FI;
    ENDFOR

```

```

        ELSE    DEST[j] ← 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**PCMPGTD (with 64-bit operands)**

```

    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
    IF DEST[63:32] > SRC[63:32]
        THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 0; FI;

```

**PCMPGTD (with 128-bit operands)**

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

```

**VPCMPGTD (VEX.128 encoded version)**

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

```

**VPCMPGTD (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_DWORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_DWORDS_GREATER(SRC1[255:128], SRC2[255:128])
DEST[MAXVL-1:256] ← 0

```

**VPCMPGTD (EVEX encoded versions)**

```

(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] > SRC2[31:0];
            ELSE CMP ← SRC1[i+31:i] > SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
            ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPCMPGTB __mmask64 __mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 __mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 __mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 __mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 __mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 __mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 __mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTD __mmask16 __mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 __mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTD __mmask8 __mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 __mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTD __mmask8 __mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTW __mmask32 __mm512_cmpgt_epi16_mask(__m512i a, __m512i b);
VPCMPGTW __mmask32 __mm512_mask_cmpgt_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPGTW __mmask16 __mm256_cmpgt_epi16_mask(__m256i a, __m256i b);
VPCMPGTW __mmask16 __mm256_mask_cmpgt_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPGTW __mmask8 __mm_cmpgt_epi16_mask(__m128i a, __m128i b);
VPCMPGTW __mmask8 __mm_mask_cmpgt_epi16_mask(__mmask8 k, __m128i a, __m128i b);
PCMPGTB: __m64 __mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW: __m64 __mm_pcmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD: __m64 __mm_pcmpgt_pi32 (__m64 m1, __m64 m2)
(V)PCMPGTB: __m128i __mm_cmpgt_epi8 (__m128i a, __m128i b)
(V)PCMPGTW: __m128i __mm_cmpgt_epi16 (__m128i a, __m128i b)
(V)DCMPGTD: __m128i __mm_cmpgt_epi32 (__m128i a, __m128i b)
VPCMPGTB:      __m256i __mm256_cmpgt_epi8 (__m256i a, __m256i b)
VPCMPGTW:      __m256i __mm256_cmpgt_epi16 (__m256i a, __m256i b)
VPCMPGTD:      __m256i __mm256_cmpgt_epi32 (__m256i a, __m256i b)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPGTD, see Exceptions Type E4.

EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb.

## PCMPGTQ — Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 /r PCMPGTQ <i>xmm1, xmm2/m128</i>	A	V/V	SSE4_2	Compare packed signed qwords in <i>xmm2/m128</i> and <i>xmm1</i> for greater than.
VEX.NDS.128.66.0F38.WIG 37 /r VPCMPGTQ <i>xmm1, xmm2, xmm3/m128</i>	B	V/V	AVX	Compare packed signed qwords in <i>xmm2</i> and <i>xmm3/m128</i> for greater than.
VEX.NDS.256.66.0F38.WIG 37 /r VPCMPGTQ <i>ymm1, ymm2, ymm3/m256</i>	B	V/V	AVX2	Compare packed signed qwords in <i>ymm2</i> and <i>ymm3/m256</i> for greater than.
EVEX.NDS.128.66.0F38.W1 37 /r VPCMPGTQ <i>k1 {k2}, xmm2, xmm3/m128/m64bcst</i>	C	V/V	AVX512VL AVX512F	Compare Greater between int64 vector <i>xmm2</i> and int64 vector <i>xmm3/m128/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F38.W1 37 /r VPCMPGTQ <i>k1 {k2}, ymm2, ymm3/m256/m64bcst</i>	C	V/V	AVX512VL AVX512F	Compare Greater between int64 vector <i>ymm2</i> and int64 vector <i>ymm3/m256/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F38.W1 37 /r VPCMPGTQ <i>k1 {k2}, zmm2, zmm3/m512/m64bcst</i>	C	V/V	AVX512F	Compare Greater between int64 vector <i>zmm2</i> and int64 vector <i>zmm3/m512/m64bcst</i> , and set vector mask <i>k1</i> to reflect the zero/nonzero status of each element of the result, under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTD/Q: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask *k2*.



**Operation****COMPARE\_QWORDS\_GREATER (SRC1, SRC2)**

```

IF SRC1[63:0] > SRC2[63:0]
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0; FI;
IF SRC1[127:64] > SRC2[127:64]
THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[127:64] ← 0; FI;

```

**VPCMPGTQ (VEX.128 encoded version)**

```

DEST[127:0] ← COMPARE_QWORDS_GREATER(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

```

**VPCMPGTQ (VEX.256 encoded version)**

```

DEST[127:0] ← COMPARE_QWORDS_GREATER(SRC1[127:0], SRC2[127:0])
DEST[255:128] ← COMPARE_QWORDS_GREATER(SRC1[255:128], SRC2[255:128])
DEST[MAXVL-1:256] ← 0

```

**VPCMPGTQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k2[j] OR \*no writemask\*

    THEN

      /\* signed comparison \*/

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN CMP ← SRC1[i+63:i] > SRC2[63:0];

        ELSE CMP ← SRC1[i+63:i] > SRC2[i+63:i];

      FI;

      IF CMP = TRUE

        THEN DEST[j] ← 1;

        ELSE DEST[j] ← 0; FI;

    ELSE     DEST[j] ← 0                   ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPCMPGTQ __mmask8 __mm512_cmpgt_epi64_mask( __m512i a, __m512i b);
VPCMPGTQ __mmask8 __mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPGTQ __mmask8 __mm256_cmpgt_epi64_mask( __m256i a, __m256i b);
VPCMPGTQ __mmask8 __mm256_mask_cmpgt_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTQ __mmask8 __mm_cmpgt_epi64_mask( __m128i a, __m128i b);
VPCMPGTQ __mmask8 __mm_mask_cmpgt_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)VPCMPGTQ:   __m128i __mm_cmpgt_epi64(__m128i a, __m128i b)
VPCMPGTQ:     __m256i __mm256_cmpgt_epi64( __m256i a, __m256i b);

```

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPCMPGTQ, see Exceptions Type E4.

## PCMPISTRI — Packed Compare Implicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 63 /r imm8 PCMPISTRI <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A.WIG 63 /r ib VPCMPISTRI <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

### Description

The instruction compares data from two strings based on the encoded value in the Imm8 Control Byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPSTRI / PCMPSTRM / PCMPISTRI / PCMPISTRM”), and generates an index stored to ECX.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise  
 ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise  
 SFlag – Set if any byte/word of xmm1 is null, reset otherwise  
 OFlag –IntRes2[0]  
 AFlag – Reset  
 PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	ECX
32 bit	xmm	xmm/m128	ECX
64 bit	xmm	xmm/m128	ECX

### Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

```
int __mm_cmpistri (__m128i a, __m128i b, const int mode);
```

### Intel C/C++ Compiler Intrinsics For Reading EFlag Results

```
int  _mm_cmpistra (__m128i a, __m128i b, const int mode);
int  _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int  _mm_cmpistro (__m128i a, __m128i b, const int mode);
int  _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int  _mm_cmpistrz (__m128i a, __m128i b, const int mode);
```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD                      If VEX.L = 1.  
                           If VEX.vvvv ≠ 1111B.

## PCMPISTRM — Packed Compare Implicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 62 /r imm8 PCMPISTRM <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating a mask, and storing the result in <i>XMM0</i> .
VEX.128.66.0F3A.WIG 62 /r ib VPCMPISTRM <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in <i>XMM0</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

### Description

The instruction compares data from two strings based on the encoded value in the imm8 byte (see Section 4.1, “Imm8 Control Byte Operation for PCMPSTR / PCMPSTRM / PCMPISTR / PCMPISTRM”) generating a mask stored to XMM0.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operation are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise  
 ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise  
 SFlag – Set if any byte/word of xmm1 is null, reset otherwise  
 OFlag – IntRes2[0]  
 AFlag – Reset  
 PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	XMM0
32 bit	xmm	xmm/m128	XMM0
64 bit	xmm	xmm/m128	XMM0

### Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

`__m128i __mm_cmpistrm (__m128i a, __m128i b, const int mode);`

### Intel C/C++ Compiler Intrinsics For Reading EFlag Results

```
int  _mm_cmpistra (__m128i a, __m128i b, const int mode);
int  _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int  _mm_cmpistro (__m128i a, __m128i b, const int mode);
int  _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int  _mm_cmpistrz (__m128i a, __m128i b, const int mode);
```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD                    If VEX.L = 1.  
                          If VEX.vvvv ≠ 1111B.

PDEP — Parallel Bits Deposit

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F2.0F38.W0 F5 /r PDEP <i>r32a</i> , <i>r32b</i> , <i>r/m32</i>	RVM	V/V	BMI2	Parallel deposit of bits from <i>r32b</i> using mask in <i>r/m32</i> , result is written to <i>r32a</i> .
VEX.NDS.LZ.F2.0F38.W1 F5 /r PDEP <i>r64a</i> , <i>r64b</i> , <i>r/m64</i>	RVM	V/N.E.	BMI2	Parallel deposit of bits from <i>r64b</i> using mask in <i>r/m64</i> , result is written to <i>r64a</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PDEP uses a mask in the second source operand (the third operand) to transfer/scatter contiguous low order bits in the first source operand (the second operand) into the destination (the first operand). PDEP takes the low bits from the first source operand and deposit them in the destination operand at the corresponding bit locations that are set in the second source operand (mask). All other bits (bits not set in mask) in destination are set to zero.

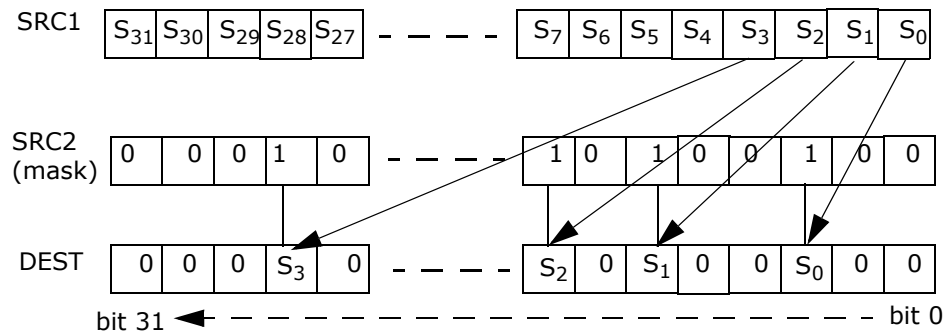


Figure 4-8. PDEP Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0;
m ← 0, k ← 0;
DO WHILE m < OperandSize
    IF MASK[ m ] = 1 THEN
        DEST[ m ] ← TEMP[ k ];
        k ← k + 1;
    FI
    m ← m + 1;
OD
```

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

PDEP:        unsigned \_\_int32 \_pdep\_u32(unsigned \_\_int32 src, unsigned \_\_int32 mask);

PDEP:        unsigned \_\_int64 \_pdep\_u64(unsigned \_\_int64 src, unsigned \_\_int32 mask);

## SIMD Floating-Point Exceptions

None

## Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD                      If VEX.W = 1.



PEXT — Parallel Bits Extract

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F3.0F38.W0 F5 /r PEXT <i>r32a</i> , <i>r32b</i> , <i>r/m32</i>	RVM	V/V	BMI2	Parallel extract of bits from <i>r32b</i> using mask in <i>r/m32</i> , result is written to <i>r32a</i> .
VEX.NDS.LZ.F3.0F38.W1 F5 /r PEXT <i>r64a</i> , <i>r64b</i> , <i>r/m64</i>	RVM	V/N.E.	BMI2	Parallel extract of bits from <i>r64b</i> using mask in <i>r/m64</i> , result is written to <i>r64a</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed.

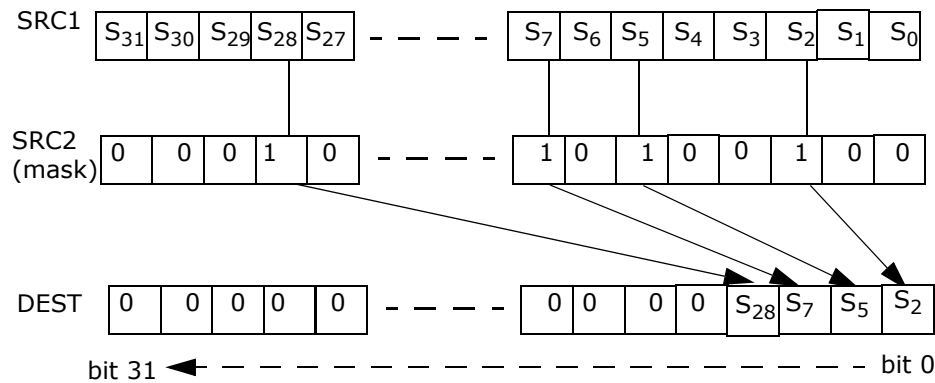


Figure 4-9. PEXT Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0;
m ← 0, k ← 0;
DO WHILE m < OperandSize
    IF MASK[ m ] = 1 THEN
        DEST[ k ] ← TEMP[ m ];
        k ← k + 1;
    FI
    m ← m + 1;
```

$$m \leftarrow m + 1;$$

OD

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

PEXT:        unsigned \_\_int32 \_pext\_u32(unsigned \_\_int32 src, unsigned \_\_int32 mask);

PEXT:        unsigned \_\_int64 \_pext\_u64(unsigned \_\_int64 src, unsigned \_\_int32 mask);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
 #UD                      If VEX.W = 1.

## PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB <i>reg/m8, xmm2, imm8</i>	A	V/V	SSE4_1	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of r32 or r64 are zeroed.
66 0F 3A 16 /r ib PEXTRD <i>r/m32, xmm2, imm8</i>	A	V/V	SSE4_1	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r/m32</i> .
66 REX.W 0F 3A 16 /r ib PEXTRQ <i>r/m64, xmm2, imm8</i>	A	V/N.E.	SSE4_1	Extract a qword integer value from <i>xmm2</i> at the source qword offset specified by <i>imm8</i> into <i>r/m64</i> .
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB <i>reg/m8, xmm2, imm8</i>	A	V <sup>1</sup> /V	AVX	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD <i>r32/m32, xmm2, imm8</i>	A	V/V	AVX	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r32/m32</i> .
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ <i>r64/m64, xmm2, imm8</i>	A	V/I <sup>2</sup>	AVX	Extract a qword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r64/m64</i> .
EVEX.128.66.0F3A.WIG 14 /r ib VPEXTRB <i>reg/m8, xmm2, imm8</i>	B	V/V	AVX512BW	Extract a byte integer value from <i>xmm2</i> at the source byte offset specified by <i>imm8</i> into <i>reg</i> or <i>m8</i> . The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.W0 16 /r ib VPEXTRD <i>r32/m32, xmm2, imm8</i>	B	V/V	AVX512DQ	Extract a dword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r32/m32</i> .
EVEX.128.66.0F3A.W1 16 /r ib VPEXTRQ <i>r64/m64, xmm2, imm8</i>	B	V/N.E. <sup>2</sup>	AVX512DQ	Extract a qword integer value from <i>xmm2</i> at the source dword offset specified by <i>imm8</i> into <i>r64/m64</i> .

## NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

## Description

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from *imm8*[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L'L must be

0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

## Operation

CASE of

```

PEXTRB: SEL ← COUNT[3:0];
        TEMP ← (Src >> SEL*8) AND FFH;
        IF (DEST = Mem8)
            THEN
                Mem8 ← TEMP[7:0];
            ELSE IF (64-Bit Mode and 64-bit register selected)
                THEN
                    R64[7:0] ← TEMP[7:0];
                    r64[63:8] ← ZERO_FILL; };
            ELSE
                R32[7:0] ← TEMP[7:0];
                r32[31:8] ← ZERO_FILL; };
        FI;
PEXTRD: SEL ← COUNT[1:0];
        TEMP ← (Src >> SEL*32) AND FFFF_FFFFH;
        DEST ← TEMP;
PEXTRQ: SEL ← COUNT[0];
        TEMP ← (Src >> SEL*64);
        DEST ← TEMP;

```

EASC:

## VPEXTRTD/VPEXTRQ

IF (64-Bit Mode and 64-bit dest operand)

THEN

```

    Src_Offset ← Imm8[0]
    r64/m64 ← (Src >> Src_Offset * 64)

```

ELSE

```

    Src_Offset ← Imm8[1:0]
    r32/m32 ← ((Src >> Src_Offset * 32) AND 0FFFFFFFH);

```

FI

## VPEXTRB ( dest=m8)

SRC\_Offset ← Imm8[3:0]

Mem8 ← (Src >> SRC\_Offset\*8)

## VPEXTRB ( dest=reg)

IF (64-Bit Mode )

THEN

```

    SRC_Offset ← Imm8[3:0]
    DEST[7:0] ← ((Src >> SRC_Offset*8) AND 0FFh)
    DEST[63:8] ← ZERO_FILL;

```

ELSE

```

    SRC_Offset ← Imm8[3:0];
    DEST[7:0] ← ((Src >> SRC_Offset*8) AND 0FFh);
    DEST[31:8] ← ZERO_FILL;

```

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

PEXTRB:        `int _mm_extract_epi8 (__m128i src, const int ndx);`  
 PEXTRD:        `int _mm_extract_epi32 (__m128i src, const int ndx);`  
 PEXTRQ:        `__int64 _mm_extract_epi64 (__m128i src, const int ndx);`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD                If VEX.L = 1 or EVEX.L'L > 0.  
                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C5 /r ib <sup>1</sup> PEXTRW <i>reg, mm, imm8</i>	A	V/V	SSE	Extract the word specified by <i>imm8</i> from <i>mm</i> and move it to <i>reg</i> , bits 15:0. The upper bits of <i>r32</i> or <i>r64</i> is zeroed.
66 0F C5 /r ib PEXTRW <i>reg, xmm, imm8</i>	A	V/V	SSE2	Extract the word specified by <i>imm8</i> from <i>xmm</i> and move it to <i>reg</i> , bits 15:0. The upper bits of <i>r32</i> or <i>r64</i> is zeroed.
66 0F 3A 15 /r ib PEXTRW <i>reg/m16, xmm, imm8</i>	B	V/V	SSE4_1	Extract the word specified by <i>imm8</i> from <i>xmm</i> and copy it to lowest 16 bits of <i>reg</i> or <i>m16</i> . Zero-extend the result in the destination, <i>r32</i> or <i>r64</i> .
VEX.128.66.0F.W0 C5 /r ib VPEXTRW <i>reg, xmm1, imm8</i>	A	V <sup>2</sup> /V	AVX	Extract the word specified by <i>imm8</i> from <i>xmm1</i> and move it to <i>reg</i> , bits 15:0. Zero-extend the result. The upper bits of <i>r64/r32</i> is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW <i>reg/m16, xmm2, imm8</i>	B	V/V	AVX	Extract a word integer value from <i>xmm2</i> at the source word offset specified by <i>imm8</i> into <i>reg</i> or <i>m16</i> . The upper bits of <i>r64/r32</i> is filled with zeros.
EVEX.128.66.0F.WIG C5 /r ib VPEXTRW <i>reg, xmm1, imm8</i>	A	V/V	AVX512B W	Extract the word specified by <i>imm8</i> from <i>xmm1</i> and move it to <i>reg</i> , bits 15:0. Zero-extend the result. The upper bits of <i>r64/r32</i> is filled with zeros.
EVEX.128.66.0F3A.WIG 15 /r ib VPEXTRW <i>reg/m16, xmm2, imm8</i>	C	V/V	AVX512B W	Extract a word integer value from <i>xmm2</i> at the source word offset specified by <i>imm8</i> into <i>reg</i> or <i>m16</i> . The upper bits of <i>r64/r32</i> is filled with zeros.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.
2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
C	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

### Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

## Operation

```
IF (DEST = Mem16)
THEN
    SEL ← COUNT[2:0];
    TEMP ← (Src >> SEL*16) AND FFFFH;
    Mem16 ← TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
THEN
    FOR (PEXTRW instruction with 64-bit source operand)
    { SEL ← COUNT[1:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r64[15:0] ← TEMP[15:0];
      r64[63:16] ← ZERO_FILL; };
    FOR (PEXTRW instruction with 128-bit source operand)
    { SEL ← COUNT[2:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r64[15:0] ← TEMP[15:0];
      r64[63:16] ← ZERO_FILL; }
ELSE
    FOR (PEXTRW instruction with 64-bit source operand)
    { SEL ← COUNT[1:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r32[15:0] ← TEMP[15:0];
      r32[31:16] ← ZERO_FILL; };
    FOR (PEXTRW instruction with 128-bit source operand)
    { SEL ← COUNT[2:0];
      TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
      r32[15:0] ← TEMP[15:0];
      r32[31:16] ← ZERO_FILL; };
FI;
FI;
```

### VPEXTRW (dest=m16)

```
SRC_Offset ← Imm8[2:0]
Mem16 ← (Src >> SRC_Offset*16)
```

**VPEXTRW ( dest=reg)**

IF (64-Bit Mode )

THEN

SRC\_Offset  $\leftarrow$  Imm8[2:0]DEST[15:0]  $\leftarrow$  ((Src >> SRC\_Offset\*16) AND 0FFFFh)DEST[63:16]  $\leftarrow$  ZERO\_FILL;

ELSE

SRC\_Offset  $\leftarrow$  Imm8[2:0]DEST[15:0]  $\leftarrow$  ((Src >> SRC\_Offset\*16) AND 0FFFFh)DEST[31:16]  $\leftarrow$  ZERO\_FILL;

FI

**Intel C/C++ Compiler Intrinsic Equivalent**

PEXTRW: int \_mm\_extract\_pi16 (\_\_m64 a, int n)

PEXTRW: int \_mm\_extract\_epi16 (\_\_m128i a, int imm)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD If VEX.L = 1 or EVEX.L'L > 0.  
 If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.



## PHADDW/PHADD — Packed Horizontal Add

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 38 01 /r <sup>1</sup> PHADDW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to <i>mm1</i> .
66 OF 38 01 /r PHADDW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to <i>xmm1</i> .
NP OF 38 02 /r PHADD <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to <i>mm1</i> .
66 OF 38 02 /r PHADD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 01 /r VPHADDW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add 16-bit integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 02 /r VPHADD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Add 32-bit integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 01 /r VPHADDW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack to <i>ymm1</i> .
VEX.NDS.256.66.0F38.WIG 02 /r VPHADD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Add 32-bit signed integers horizontally, pack to <i>ymm1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PHADDW adds two adjacent 16-bit signed integers horizontally from the source and destination operands and packs the 16-bit signed results to the destination operand (first operand). (V)PHADD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Note that these instructions can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Legacy SSE instructions: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

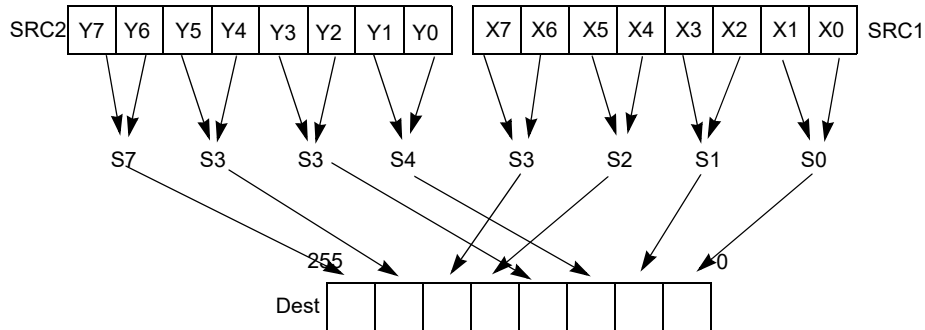


Figure 4-10. 256-bit VPHADD Instruction Operation

## Operation

### PHADDW (with 64-bit operands)

```
mm1[15:0] = mm1[31:16] + mm1[15:0];
mm1[31:16] = mm1[63:48] + mm1[47:32];
mm1[47:32] = mm2/m64[31:16] + mm2/m64[15:0];
mm1[63:48] = mm2/m64[63:48] + mm2/m64[47:32];
```

### PHADDW (with 128-bit operands)

```
xmm1[15:0] = xmm1[31:16] + xmm1[15:0];
xmm1[31:16] = xmm1[63:48] + xmm1[47:32];
xmm1[47:32] = xmm1[95:80] + xmm1[79:64];
xmm1[63:48] = xmm1[127:112] + xmm1[111:96];
xmm1[79:64] = xmm2/m128[31:16] + xmm2/m128[15:0];
xmm1[95:80] = xmm2/m128[63:48] + xmm2/m128[47:32];
xmm1[111:96] = xmm2/m128[95:80] + xmm2/m128[79:64];
xmm1[127:112] = xmm2/m128[127:112] + xmm2/m128[111:96];
```

### VPHADDW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[MAXVL-1:128] ← 0
```

**VPHADDW (VEX.256 encoded version)**

$DEST[15:0] \leftarrow SRC1[31:16] + SRC1[15:0]$   
 $DEST[31:16] \leftarrow SRC1[63:48] + SRC1[47:32]$   
 $DEST[47:32] \leftarrow SRC1[95:80] + SRC1[79:64]$   
 $DEST[63:48] \leftarrow SRC1[127:112] + SRC1[111:96]$   
 $DEST[79:64] \leftarrow SRC2[31:16] + SRC2[15:0]$   
 $DEST[95:80] \leftarrow SRC2[63:48] + SRC2[47:32]$   
 $DEST[111:96] \leftarrow SRC2[95:80] + SRC2[79:64]$   
 $DEST[127:112] \leftarrow SRC2[127:112] + SRC2[111:96]$   
 $DEST[143:128] \leftarrow SRC1[159:144] + SRC1[143:128]$   
 $DEST[159:144] \leftarrow SRC1[191:176] + SRC1[175:160]$   
 $DEST[175:160] \leftarrow SRC1[223:208] + SRC1[207:192]$   
 $DEST[191:176] \leftarrow SRC1[255:240] + SRC1[239:224]$   
 $DEST[207:192] \leftarrow SRC2[127:112] + SRC2[143:128]$   
 $DEST[223:208] \leftarrow SRC2[159:144] + SRC2[175:160]$   
 $DEST[239:224] \leftarrow SRC2[191:176] + SRC2[207:192]$   
 $DEST[255:240] \leftarrow SRC2[223:208] + SRC2[239:224]$

**PHADDD (with 64-bit operands)**

$mm1[31:0] = mm1[63:32] + mm1[31:0];$   
 $mm1[63:32] = mm2/m64[63:32] + mm2/m64[31:0];$

**PHADDD (with 128-bit operands)**

$xmm1[31:0] = xmm1[63:32] + xmm1[31:0];$   
 $xmm1[63:32] = xmm1[127:96] + xmm1[95:64];$   
 $xmm1[95:64] = xmm2/m128[63:32] + xmm2/m128[31:0];$   
 $xmm1[127:96] = xmm2/m128[127:96] + xmm2/m128[95:64];$

**VPHADDD (VEX.128 encoded version)**

$DEST[31:0] \leftarrow SRC1[63:32] + SRC1[31:0]$   
 $DEST[63:32] \leftarrow SRC1[127:96] + SRC1[95:64]$   
 $DEST[95:64] \leftarrow SRC2[63:32] + SRC2[31:0]$   
 $DEST[127:96] \leftarrow SRC2[127:96] + SRC2[95:64]$   
 $DEST[MAXVL-1:128] \leftarrow 0$

**VPHADDD (VEX.256 encoded version)**

$DEST[31:0] \leftarrow SRC1[63:32] + SRC1[31:0]$   
 $DEST[63:32] \leftarrow SRC1[127:96] + SRC1[95:64]$   
 $DEST[95:64] \leftarrow SRC2[63:32] + SRC2[31:0]$   
 $DEST[127:96] \leftarrow SRC2[127:96] + SRC2[95:64]$   
 $DEST[159:128] \leftarrow SRC1[191:160] + SRC1[159:128]$   
 $DEST[191:160] \leftarrow SRC1[255:224] + SRC1[223:192]$   
 $DEST[223:192] \leftarrow SRC2[191:160] + SRC2[159:128]$   
 $DEST[255:224] \leftarrow SRC2[255:224] + SRC2[223:192]$

**Intel C/C++ Compiler Intrinsic Equivalents**

PHADDW: `__m64 _mm_hadd_pi16 (__m64 a, __m64 b)`  
 PHADDD: `__m64 _mm_hadd_pi32 (__m64 a, __m64 b)`  
 (V)PHADDW: `__m128i _mm_hadd_epi16 (__m128i a, __m128i b)`  
 (V)PHADDD: `__m128i _mm_hadd_epi32 (__m128i a, __m128i b)`  
 VPHADDW: `__m256i _mm256_hadd_epi16 (__m256i a, __m256i b)`  
 VPHADDD: `__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

## PHADDSW — Packed Horizontal Add and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 03 /r <sup>1</sup> PHADDSW <i>mm1, mm2/m64</i>	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to <i>mm1</i> .
66 0F 38 03 /r PHADDSW <i>xmm1, xmm2/m128</i>	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 03 /r VPHADDSW <i>xmm1, xmm2, xmm3/m128</i>	RVM	V/V	AVX	Add 16-bit signed integers horizontally, pack saturated integers to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 03 /r VPHADDSW <i>ymm1, ymm2, ymm3/m256</i>	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack saturated integers to <i>ymm1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PHADDSW adds two adjacent signed 16-bit integers horizontally from the source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PHADDSW (with 64-bit operands)

```
mm1[15-0] = SaturateToSignedWord((mm1[31-16] + mm1[15-0]);
mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);
```

**PHADDSW (with 128-bit operands)**

```

xmm1[15:0] = SaturateToSignedWord(xmm1[31:16] + xmm1[15:0]);
xmm1[31:16] = SaturateToSignedWord(xmm1[63:48] + xmm1[47:32]);
xmm1[47:32] = SaturateToSignedWord(xmm1[95:80] + xmm1[79:64]);
xmm1[63:48] = SaturateToSignedWord(xmm1[127:112] + xmm1[111:96]);
xmm1[79:64] = SaturateToSignedWord(xmm2/m128[31:16] + xmm2/m128[15:0]);
xmm1[95:80] = SaturateToSignedWord(xmm2/m128[63:48] + xmm2/m128[47:32]);
xmm1[111:96] = SaturateToSignedWord(xmm2/m128[95:80] + xmm2/m128[79:64]);
xmm1[127:112] = SaturateToSignedWord(xmm2/m128[127:112] + xmm2/m128[111:96]);

```

**VPHADDSW (VEX.128 encoded version)**

```

DEST[15:0] = SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[MAXVL-1:128] ← 0

```

**VPHADDSW (VEX.256 encoded version)**

```

DEST[15:0] = SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[143:128] = SaturateToSignedWord(SRC1[159:144] + SRC1[143:128])
DEST[159:144] = SaturateToSignedWord(SRC1[191:176] + SRC1[175:160])
DEST[175:160] = SaturateToSignedWord(SRC1[223:208] + SRC1[207:192])
DEST[191:176] = SaturateToSignedWord(SRC1[255:240] + SRC1[239:224])
DEST[207:192] = SaturateToSignedWord(SRC2[127:112] + SRC2[143:128])
DEST[223:208] = SaturateToSignedWord(SRC2[159:144] + SRC2[175:160])
DEST[239:224] = SaturateToSignedWord(SRC2[191:160] + SRC2[159:128])
DEST[255:240] = SaturateToSignedWord(SRC2[255:240] + SRC2[239:224])

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

PHADDSW:    __m64 _mm_hadds_pi16 (__m64 a, __m64 b)
(V)PHADDSW: __m128i _mm_hadds_epi16 (__m128i a, __m128i b)
VPHADDSW:   __m256i _mm256_hadds_epi16 (__m256i a, __m256i b)

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

## PHMINPOSUW — Packed Horizontal Word Minimum

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 41 /r PHMINPOSUW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Find the minimum unsigned word in <i>xmm2/m128</i> and place its value in the low word of <i>xmm1</i> and its index in the second-lowest word of <i>xmm1</i> .
VEX.128.66.0F38.WIG 41 /r VPHMINPOSUW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Find the minimum unsigned word in <i>xmm2/m128</i> and place its value in the low word of <i>xmm1</i> and its index in the second-lowest word of <i>xmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Determine the minimum unsigned word value in the source operand (second operand) and place the unsigned word in the low word (bits 0-15) of the destination operand (first operand). The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding XMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination XMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PHMINPOSUW (128-bit Legacy SSE version)

INDEX  $\leftarrow$  0;

MIN  $\leftarrow$  SRC[15:0]

IF (SRC[31:16] < MIN)

THEN INDEX  $\leftarrow$  1; MIN  $\leftarrow$  SRC[31:16]; FI;

IF (SRC[47:32] < MIN)

THEN INDEX  $\leftarrow$  2; MIN  $\leftarrow$  SRC[47:32]; FI;

\* Repeat operation for words 3 through 6

IF (SRC[127:112] < MIN)

THEN INDEX  $\leftarrow$  7; MIN  $\leftarrow$  SRC[127:112]; FI;

DEST[15:0]  $\leftarrow$  MIN;

DEST[18:16]  $\leftarrow$  INDEX;

DEST[127:19]  $\leftarrow$  000000000000000000000000000000H;

**VPHMINPOSUW (VEX.128 encoded version)**

$INDEX \leftarrow 0$   
 $MIN \leftarrow SRC[15:0]$   
 IF  $(SRC[31:16] < MIN)$  THEN  $INDEX \leftarrow 1$ ;  $MIN \leftarrow SRC[31:16]$   
 IF  $(SRC[47:32] < MIN)$  THEN  $INDEX \leftarrow 2$ ;  $MIN \leftarrow SRC[47:32]$   
 \* Repeat operation for words 3 through 6  
 IF  $(SRC[127:112] < MIN)$  THEN  $INDEX \leftarrow 7$ ;  $MIN \leftarrow SRC[127:112]$   
 $DEST[15:0] \leftarrow MIN$   
 $DEST[18:16] \leftarrow INDEX$   
 $DEST[127:19] \leftarrow 00000000000000000000000000000000H$   
 $DEST[MAXVL-1:128] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

PHMINPOSUW: `__m128i _mm_minpos_epu16(__m128i packed_words);`

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD	If VEX.L = 1. If VEX.vvvv $\neq$ 1111B.
-----	--



## PHSUBW/PHSUBD — Packed Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 38 05 /r <sup>1</sup> PHSUBW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to <i>mm1</i> .
66 OF 38 05 /r PHSUBW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to <i>xmm1</i> .
NP OF 38 06 /r PHSUBD <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to <i>mm1</i> .
66 OF 38 06 /r PHSUBD <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 05 /r VPHSUBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract 16-bit signed integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 06 /r VPHSUBD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract 32-bit signed integers horizontally, pack to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 05 /r VPHSUBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract 16-bit signed integers horizontally, pack to <i>ymm1</i> .
VEX.NDS.256.66.0F38.WIG 06 /r VPHSUBD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract 32-bit signed integers horizontally, pack to <i>ymm1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands, and packs the signed 16-bit results to the destination operand (first operand). (V)PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

## Operation

### PHSUBW (with 64-bit operands)

```
mm1[15-0] = mm1[15-0] - mm1[31-16];
mm1[31-16] = mm1[47-32] - mm1[63-48];
mm1[47-32] = mm2/m64[15-0] - mm2/m64[31-16];
mm1[63-48] = mm2/m64[47-32] - mm2/m64[63-48];
```

### PHSUBW (with 128-bit operands)

```
xmm1[15-0] = xmm1[15-0] - xmm1[31-16];
xmm1[31-16] = xmm1[47-32] - xmm1[63-48];
xmm1[47-32] = xmm1[79-64] - xmm1[95-80];
xmm1[63-48] = xmm1[111-96] - xmm1[127-112];
xmm1[79-64] = xmm2/m128[15-0] - xmm2/m128[31-16];
xmm1[95-80] = xmm2/m128[47-32] - xmm2/m128[63-48];
xmm1[111-96] = xmm2/m128[79-64] - xmm2/m128[95-80];
xmm1[127-112] = xmm2/m128[111-96] - xmm2/m128[127-112];
```

### VPHSUBW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[MAXVL-1:128] ← 0
```

### VPHSUBW (VEX.256 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[143:128] ← SRC1[143:128] - SRC1[159:144]
DEST[159:144] ← SRC1[175:160] - SRC1[191:176]
DEST[175:160] ← SRC1[207:192] - SRC1[223:208]
DEST[191:176] ← SRC1[239:224] - SRC1[255:240]
DEST[207:192] ← SRC2[143:128] - SRC2[159:144]
DEST[223:208] ← SRC2[175:160] - SRC2[191:176]
DEST[239:224] ← SRC2[207:192] - SRC2[223:208]
DEST[255:240] ← SRC2[239:224] - SRC2[255:240]
```

### PHSUBD (with 64-bit operands)

```
mm1[31-0] = mm1[31-0] - mm1[63-32];
mm1[63-32] = mm2/m64[31-0] - mm2/m64[63-32];
```

**PHSUBD (with 128-bit operands)**

```

xmm1[31-0] = xmm1[31-0] - xmm1[63-32];
xmm1[63-32] = xmm1[95-64] - xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] - xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] - xmm2/m128[127-96];

```

**VPHSUBD (VEX.128 encoded version)**

```

DEST[31-0] ← SRC1[31-0] - SRC1[63-32]
DEST[63-32] ← SRC1[95-64] - SRC1[127-96]
DEST[95-64] ← SRC2[31-0] - SRC2[63-32]
DEST[127-96] ← SRC2[95-64] - SRC2[127-96]
DEST[MAXVL-1:128] ← 0

```

**VPHSUBD (VEX.256 encoded version)**

```

DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC1[191:160]
DEST[191:160] ← SRC1[223:192] - SRC1[255:224]
DEST[223:192] ← SRC2[159:128] - SRC2[191:160]
DEST[255:224] ← SRC2[223:192] - SRC2[255:224]

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

PHSUBW:    __m64 _mm_hsub_pi16 (__m64 a, __m64 b)
PHSUBD:    __m64 _mm_hsub_pi32 (__m64 a, __m64 b)
(V)PHSUBW: __m128i _mm_hsub_epi16 (__m128i a, __m128i b)
(V)PHSUBD: __m128i _mm_hsub_epi32 (__m128i a, __m128i b)
VPHSUBW:   __m256i _mm256_hsub_epi16 (__m256i a, __m256i b)
VPHSUBD:   __m256i _mm256_hsub_epi32 (__m256i a, __m256i b)

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                      If VEX.L = 1.

## PHSUBSW — Packed Horizontal Subtract and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 07 /r <sup>1</sup> PHSUBSW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to <i>mm1</i> .
66 0F 38 07 /r PHSUBSW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 07 /r VPHSUBSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Subtract 16-bit signed integer horizontally, pack saturated integers to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 07 /r VPHSUBSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Subtract 16-bit signed integer horizontally, pack saturated integers to <i>ymm1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

(V)PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

### Operation

#### PHSUBSW (with 64-bit operands)

```
mm1[15-0] = SaturateToSignedWord(mm1[15-0] - mm1[31-16]);
mm1[31-16] = SaturateToSignedWord(mm1[47-32] - mm1[63-48]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[15-0] - mm2/m64[31-16]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[47-32] - mm2/m64[63-48]);
```

**PHSUBSW (with 128-bit operands)**

```

xmm1[15:0] = SaturateToSignedWord(xmm1[15:0] - xmm1[31:16]);
xmm1[31:16] = SaturateToSignedWord(xmm1[47:32] - xmm1[63:48]);
xmm1[47:32] = SaturateToSignedWord(xmm1[79:64] - xmm1[95:80]);
xmm1[63:48] = SaturateToSignedWord(xmm1[111:96] - xmm1[127:112]);
xmm1[79:64] = SaturateToSignedWord(xmm2/m128[15:0] - xmm2/m128[31:16]);
xmm1[95:80] = SaturateToSignedWord(xmm2/m128[47:32] - xmm2/m128[63:48]);
xmm1[111:96] = SaturateToSignedWord(xmm2/m128[79:64] - xmm2/m128[95:80]);
xmm1[127:112] = SaturateToSignedWord(xmm2/m128[111:96] - xmm2/m128[127:112]);

```

**VPHSUBSW (VEX.128 encoded version)**

```

DEST[15:0] = SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[MAXVL-1:128] ← 0

```

**VPHSUBSW (VEX.256 encoded version)**

```

DEST[15:0] = SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[143:128] = SaturateToSignedWord(SRC1[143:128] - SRC1[159:144])
DEST[159:144] = SaturateToSignedWord(SRC1[175:160] - SRC1[191:176])
DEST[175:160] = SaturateToSignedWord(SRC1[207:192] - SRC1[223:208])
DEST[191:176] = SaturateToSignedWord(SRC1[239:224] - SRC1[255:240])
DEST[207:192] = SaturateToSignedWord(SRC2[143:128] - SRC2[159:144])
DEST[223:208] = SaturateToSignedWord(SRC2[175:160] - SRC2[191:176])
DEST[239:224] = SaturateToSignedWord(SRC2[207:192] - SRC2[223:208])
DEST[255:240] = SaturateToSignedWord(SRC2[239:224] - SRC2[255:240])

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

PHSUBSW:      __m64 _mm_hsubs_pi16 (__m64 a, __m64 b)
(V)PHSUBSW:   __m128i _mm_hsubs_epi16 (__m128i a, __m128i b)
VPHSUBSW:     __m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

## PINSRB/PINSRD/PINSRQ — Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB <i>xmm1</i> , <i>r32/m8</i> , <i>imm8</i>	A	V/V	SSE4_1	Insert a byte integer value from <i>r32/m8</i> into <i>xmm1</i> at the destination element in <i>xmm1</i> specified by <i>imm8</i> .
66 0F 3A 22 /r ib PINSRD <i>xmm1</i> , <i>r/m32</i> , <i>imm8</i>	A	V/V	SSE4_1	Insert a dword integer value from <i>r/m32</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
66 REX.W 0F 3A 22 /r ib PINSRQ <i>xmm1</i> , <i>r/m64</i> , <i>imm8</i>	A	V/N. E.	SSE4_1	Insert a qword integer value from <i>r/m64</i> into the <i>xmm1</i> at the destination element specified by <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB <i>xmm1</i> , <i>xmm2</i> , <i>r32/m8</i> , <i>imm8</i>	B	V <sup>1</sup> /V	AVX	Merge a byte integer value from <i>r32/m8</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the byte offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD <i>xmm1</i> , <i>xmm2</i> , <i>r/m32</i> , <i>imm8</i>	B	V/V	AVX	Insert a dword integer value from <i>r32/m32</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the dword offset in <i>imm8</i> .
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ <i>xmm1</i> , <i>xmm2</i> , <i>r/m64</i> , <i>imm8</i>	B	V/I <sup>2</sup>	AVX	Insert a qword integer value from <i>r64/m64</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the qword offset in <i>imm8</i> .
EVEX.NDS.128.66.0F3A.WIG 20 /r ib VPINSRB <i>xmm1</i> , <i>xmm2</i> , <i>r32/m8</i> , <i>imm8</i>	C	V/V	AVX512BW	Merge a byte integer value from <i>r32/m8</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the byte offset in <i>imm8</i> .
EVEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD <i>xmm1</i> , <i>xmm2</i> , <i>r32/m32</i> , <i>imm8</i>	C	V/V	AVX512DQ	Insert a dword integer value from <i>r32/m32</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the dword offset in <i>imm8</i> .
EVEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ <i>xmm1</i> , <i>xmm2</i> , <i>r64/m64</i> , <i>imm8</i>	C	V/N.E. <sup>2</sup>	AVX512DQ	Insert a qword integer value from <i>r64/m64</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the qword offset in <i>imm8</i> .

### NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

## Operation

CASE OF

```

PINSRB: SEL ← COUNT[3:0];
        MASK ← (OFFH << (SEL * 8));
        TEMP ← (((SRC[7:0] << (SEL * 8)) AND MASK);
PINSRD: SEL ← COUNT[1:0];
        MASK ← (OFFFFFFFFFH << (SEL * 32));
        TEMP ← (((SRC << (SEL * 32)) AND MASK) ;
PINSRQ: SEL ← COUNT[0]
        MASK ← (FFFFFFFFFFFFFFFFFH << (SEL * 64));
        TEMP ← (((SRC << (SEL * 64)) AND MASK) ;

```

ESAC;

DEST ← ((DEST AND NOT MASK) OR TEMP);

### VPINSRB (VEX/EVEX encoded version)

```

SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0

```

### VPINSRD (VEX/EVEX encoded version)

```

SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0

```

### VPINSRQ (VEX/EVEX encoded version)

```

SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0

```

## Intel C/C++ Compiler Intrinsic Equivalent

```

PINSRB:    __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);
PINSRD:    __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ:    __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);

```

## Flags Affected

None.

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD                      If VEX.L = 1 or EVEX.L'L > 0.



## PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF C4 /r ib <sup>1</sup> PINSRW <i>mm</i> , <i>r32/m16</i> , <i>imm8</i>	A	V/V	SSE	Insert the low word from <i>r32</i> or from <i>m16</i> into <i>mm</i> at the word position specified by <i>imm8</i> .
66 OF C4 /r ib PINSRW <i>xmm</i> , <i>r32/m16</i> , <i>imm8</i>	A	V/V	SSE2	Move the low word of <i>r32</i> or from <i>m16</i> into <i>xmm</i> at the word position specified by <i>imm8</i> .
VEX.NDS.128.66.OF.W0 C4 /r ib VPINSRW <i>xmm1</i> , <i>xmm2</i> , <i>r32/m16</i> , <i>imm8</i>	B	V <sup>2</sup> /V	AVX	Insert a word integer value from <i>r32/m16</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the word offset in <i>imm8</i> .
EVEX.NDS.128.66.OF.WIG C4 /r ib VPINSRW <i>xmm1</i> , <i>xmm2</i> , <i>r32/m16</i> , <i>imm8</i>	C	V/V	AVX512BW	Insert a word integer value from <i>r32/m16</i> and rest from <i>xmm2</i> into <i>xmm1</i> at the word offset in <i>imm8</i> .

### NOTES:

- See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.
- In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

## Operation

### PINSRW (with 64-bit source operand)

```
SEL ← COUNT AND 3H;
CASE (Determine word position) OF
  SEL ← 0:   MASK ← 000000000000FFFFH;
  SEL ← 1:   MASK ← 00000000FFFF0000H;
  SEL ← 2:   MASK ← 0000FFFF00000000H;
  SEL ← 3:   MASK ← FFFF000000000000H;
DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);
```

### PINSRW (with 128-bit source operand)

```
SEL ← COUNT AND 7H;
CASE (Determine word position) OF
  SEL ← 0:   MASK ← 000000000000000000000000FFFFH;
  SEL ← 1:   MASK ← 000000000000000000000000FFFF0000H;
  SEL ← 2:   MASK ← 000000000000000000000000FFFF00000000H;
  SEL ← 3:   MASK ← 000000000000000000000000FFFF000000000000H;
  SEL ← 4:   MASK ← 000000000000000000000000FFFF000000000000H;
  SEL ← 5:   MASK ← 00000000FFFF000000000000000000000000H;
  SEL ← 6:   MASK ← 0000FFFF0000000000000000000000000000H;
  SEL ← 7:   MASK ← FFFF00000000000000000000000000000000H;
DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);
```

### VPINSRW (VEX/EVEX encoded version)

```
SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0
```

## Intel C/C++ Compiler Intrinsic Equivalent

```
PINSRW:    __m64 _mm_insert_pi16 (__m64 a, int d, int n)
PINSRW:    __m128i _mm_insert_epi16 (__m128i a, int b, int imm)
```

## Flags Affected

None.

## Numeric Exceptions

None.

## Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;  
 EVEX-encoded instruction, see Exceptions Type E9NF.  
 #UD If VEX.L = 1 or EVEX.L'L > 0.

## PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 04 /r <sup>1</sup> PMADDUBSW <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>mm1</i> .
66 0F 38 04 /r PMADDUBSW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>ymm1</i> .
EVEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F38.WIG 04 /r VPMADDUBSW <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
B	NA	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
C	Full Mem	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

### Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

## Operation

### PMADDUBSW (with 64 bit operands)

```
DEST[15:0] = SaturateToSignedWord(SRC[15:8]*DEST[15:8]+SRC[7:0]*DEST[7:0]);
DEST[31:16] = SaturateToSignedWord(SRC[31:24]*DEST[31:24]+SRC[23:16]*DEST[23:16]);
DEST[47:32] = SaturateToSignedWord(SRC[47:40]*DEST[47:40]+SRC[39:32]*DEST[39:32]);
DEST[63:48] = SaturateToSignedWord(SRC[63:56]*DEST[63:56]+SRC[55:48]*DEST[55:48]);
```

### PMADDUBSW (with 128 bit operands)

```
DEST[15:0] = SaturateToSignedWord(SRC[15:8]* DEST[15:8]+SRC[7:0]*DEST[7:0]);
// Repeat operation for 2nd through 7th word
SRC1/DEST[127:112] = SaturateToSignedWord(SRC[127:120]*DEST[127:120]+ SRC[119:112]* DEST[119:112]);
```

### VPADDUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[MAXVL-1:128] ← 0
```

### VPADDUBSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[MAXVL-1:256] ← 0
```

### VPADDUBSW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateToSignedWord(SRC2[i+15:i+8]* SRC1[i+15:i+8] + SRC2[i+7:i]*SRC1[i+7:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMADDUBSW __m512i _mm512_mddubs_epi16(__m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_mask_mddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_maskz_mddubs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i _mm256_mask_mddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i _mm256_maskz_mddubs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i _mm_mask_mddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i _mm_maskz_mddubs_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDUBSW: __m64 _mm_maddubs_pi16(__m64 a, __m64 b)
(V)PMADDUBSW: __m128i _mm_maddubs_epi16(__m128i a, __m128i b)
VPMADDUBSW:      __m256i _mm256_maddubs_epi16(__m256i a, __m256i b)

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F5 /r <sup>1</sup> PMADDWD <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Multiply the packed words in <i>mm</i> by the packed words in <i>mm/m64</i> , add adjacent doubleword results, and store in <i>mm</i> .
66 0F F5 /r PMADDWD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Multiply the packed word integers in <i>xmm1</i> by the packed word integers in <i>xmm2/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG F5 /r VPMADDWD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Multiply the packed word integers in <i>xmm2</i> by the packed word integers in <i>xmm3/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> .
VEX.NDS.256.66.0F.WIG F5 /r VPMADDWD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Multiply the packed word integers in <i>ymm2</i> by the packed word integers in <i>ymm3/m256</i> , add adjacent doubleword results, and store in <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG F5 /r VPMADDWD <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Multiply the packed word integers in <i>xmm2</i> by the packed word integers in <i>xmm3/m128</i> , add adjacent doubleword results, and store in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG F5 /r VPMADDWD <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Multiply the packed word integers in <i>ymm2</i> by the packed word integers in <i>ymm3/m256</i> , add adjacent doubleword results, and store in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG F5 /r VPMADDWD <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Multiply the packed word integers in <i>zmm2</i> by the packed word integers in <i>zmm3/m512</i> , add adjacent doubleword results, and store in <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
B	NA	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
C	Full Mem	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

### Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-11 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

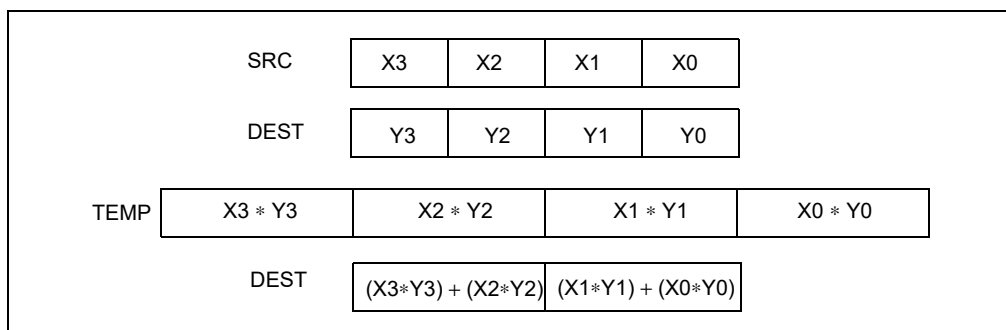
Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.



**Figure 4-11. PMADDWD Execution Model Using 64-bit Operands**

## Operation

### PMADDWD (with 64-bit operands)

$$\text{DEST}[31:0] \leftarrow (\text{DEST}[15:0] * \text{SRC}[15:0]) + (\text{DEST}[31:16] * \text{SRC}[31:16]);$$

$$\text{DEST}[63:32] \leftarrow (\text{DEST}[47:32] * \text{SRC}[47:32]) + (\text{DEST}[63:48] * \text{SRC}[63:48]);$$

### PMADDWD (with 128-bit operands)

$$\text{DEST}[31:0] \leftarrow (\text{DEST}[15:0] * \text{SRC}[15:0]) + (\text{DEST}[31:16] * \text{SRC}[31:16]);$$

$$\text{DEST}[63:32] \leftarrow (\text{DEST}[47:32] * \text{SRC}[47:32]) + (\text{DEST}[63:48] * \text{SRC}[63:48]);$$

$$\text{DEST}[95:64] \leftarrow (\text{DEST}[79:64] * \text{SRC}[79:64]) + (\text{DEST}[95:80] * \text{SRC}[95:80]);$$

$$\text{DEST}[127:96] \leftarrow (\text{DEST}[111:96] * \text{SRC}[111:96]) + (\text{DEST}[127:112] * \text{SRC}[127:112]);$$

### VPMADDWD (VEX.128 encoded version)

$$\text{DEST}[31:0] \leftarrow (\text{SRC1}[15:0] * \text{SRC2}[15:0]) + (\text{SRC1}[31:16] * \text{SRC2}[31:16])$$

$$\text{DEST}[63:32] \leftarrow (\text{SRC1}[47:32] * \text{SRC2}[47:32]) + (\text{SRC1}[63:48] * \text{SRC2}[63:48])$$

$$\text{DEST}[95:64] \leftarrow (\text{SRC1}[79:64] * \text{SRC2}[79:64]) + (\text{SRC1}[95:80] * \text{SRC2}[95:80])$$

$$\text{DEST}[127:96] \leftarrow (\text{SRC1}[111:96] * \text{SRC2}[111:96]) + (\text{SRC1}[127:112] * \text{SRC2}[127:112])$$

$$\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$$

**VPMADDWD (VEX.256 encoded version)**

$DEST[31:0] \leftarrow (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])$   
 $DEST[63:32] \leftarrow (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])$   
 $DEST[95:64] \leftarrow (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])$   
 $DEST[127:96] \leftarrow (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])$   
 $DEST[159:128] \leftarrow (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])$   
 $DEST[191:160] \leftarrow (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])$   
 $DEST[223:192] \leftarrow (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])$   
 $DEST[255:224] \leftarrow (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])$   
 $DEST[MAXVL-1:256] \leftarrow 0$

**VPMADDWD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← (SRC2[i+31:i+16]* SRC1[i+31:i+16]) + (SRC2[i+15:i]*SRC1[i+15:i])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+31:i] = 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMADDWD \_\_m512i \_\_mm512\_mdd\_epi16(\_\_m512i a, \_\_m512i b);  
VPMADDWD \_\_m512i \_\_mm512\_mask\_mdd\_epi16(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
VPMADDWD \_\_m512i \_\_mm512\_maskz\_mdd\_epi16(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
VPMADDWD \_\_m256i \_\_mm256\_mask\_mdd\_epi16(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
VPMADDWD \_\_m256i \_\_mm256\_maskz\_mdd\_epi16(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
VPMADDWD \_\_m128i \_\_mm\_mask\_mdd\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
VPMADDWD \_\_m128i \_\_mm\_maskz\_mdd\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
PMADDWD: \_\_m64 \_\_mm\_madd\_pi16(\_\_m64 m1, \_\_m64 m2)  
(V)PMADDWD: \_\_m128i \_\_mm\_madd\_epi16 (\_\_m128i a, \_\_m128i b)  
VPMADDWD: \_\_m256i \_\_mm256\_madd\_epi16 (\_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.



**PMAXSB/PMASW/PMASD/PMASQ—Maximum of Packed Signed Integers**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EE /r <sup>1</sup> PMASW mm1, mm2/m64	A	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return maximum values.
66 0F 38 3C /r PMASB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMASW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMASD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3C /r VPMASB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F.WIG EE /r VPMASW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3D /r VPMASD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3C /r VPMASB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F.WIG EE /r VPMASW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F38.WIG 3D /r VPMASD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.NDS.128.66.0F38.WIG 3C /r VPMASB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 3C /r VPMASB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 3C /r VPMASB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG EE /r VPMASW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EE /r VPMASW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG EE /r VPMASW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W0 3D /r VPMASD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F38.W0 3D /r VPMAXSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 3D /r VPMAXSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 3D /r VPMAXSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAWSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation****PMAWSW (64-bit operands)**

```

IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
IF DEST[63:48] > SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; FI;

```

**PMAWSB (128-bit Legacy SSE version)**

```

IF DEST[7:0] > SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] > SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)

```

**VPMAWSB (VEX.128 encoded version)**

```

IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0

```

**VPMAWSB (VEX.256 encoded version)**

```

IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] > SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:256] ← 0

```

**VPMASB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
            ELSE                               ; zeroing-masking
                DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**PMASW (128-bit Legacy SSE version)**

```

IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] > SRC[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)

```

**VPMASW (VEX.128 encoded version)**

```

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] > SRC2[127:112] THEN
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0

```

**VPMASW (VEX.256 encoded version)**

```

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] > SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:256] ← 0

```

**VPMAXSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\* THEN

IF SRC1[i+15:i] &gt; SRC2[i+15:i]

THEN DEST[i+15:i] ← SRC1[i+15:i];

ELSE DEST[i+15:i] ← SRC2[i+15:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**PMAXSD (128-bit Legacy SSE version)**

IF DEST[31:0] &gt; SRC[31:0] THEN

DEST[31:0] ← DEST[31:0];

ELSE

DEST[31:0] ← SRC[31:0]; FI;

(\* Repeat operation for 2nd through 7th words in source and destination operands \*)

IF DEST[127:96] &gt; SRC[127:96] THEN

DEST[127:96] ← DEST[127:96];

ELSE

DEST[127:96] ← SRC[127:96]; FI;

DEST[MAXVL-1:128] (Unmodified)

**VPMAXSD (VEX.128 encoded version)**

IF SRC1[31:0] &gt; SRC2[31:0] THEN

DEST[31:0] ← SRC1[31:0];

ELSE

DEST[31:0] ← SRC2[31:0]; FI;

(\* Repeat operation for 2nd through 3rd dwords in source and destination operands \*)

IF SRC1[127:96] &gt; SRC2[127:96] THEN

DEST[127:96] ← SRC1[127:96];

ELSE

DEST[127:96] ← SRC2[127:96]; FI;

DEST[MAXVL-1:128] ← 0

**VPMAXSD (VEX.256 encoded version)**

IF SRC1[31:0] &gt; SRC2[31:0] THEN

DEST[31:0] ← SRC1[31:0];

ELSE

DEST[31:0] ← SRC2[31:0]; FI;

(\* Repeat operation for 2nd through 7th dwords in source and destination operands \*)

IF SRC1[255:224] &gt; SRC2[255:224] THEN

DEST[255:224] ← SRC1[255:224];

ELSE

DEST[255:224] ← SRC2[255:224]; FI;

DEST[MAXVL-1:256] ← 0

**VPMAXSD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[j+31:i] > SRC2[31:0]
          THEN DEST[j+31:i] ← SRC1[j+31:i];
          ELSE DEST[j+31:i] ← SRC2[31:0];
        FI;
      ELSE
        IF SRC1[j+31:i] > SRC2[i+31:i]
          THEN DEST[j+31:i] ← SRC1[j+31:i];
          ELSE DEST[j+31:i] ← SRC2[i+31:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
        ELSE DEST[j+31:i] ← 0 ; zeroing-masking
      FI
    FI;
  ENDFOR
  DEST[MAXVL-1:VL] ← 0

```

**VPMAXSQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        IF SRC1[j+63:i] > SRC2[63:0]
          THEN DEST[j+63:i] ← SRC1[j+63:i];
          ELSE DEST[j+63:i] ← SRC2[63:0];
        FI;
      ELSE
        IF SRC1[j+63:i] > SRC2[i+63:i]
          THEN DEST[j+63:i] ← SRC1[j+63:i];
          ELSE DEST[j+63:i] ← SRC2[i+63:i];
        FI;
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
        ELSE ; zeroing-masking
          THEN DEST[j+63:i] ← 0
        FI
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
        ELSE ; zeroing-masking
          THEN DEST[j+63:i] ← 0
        FI
      FI;
    ENDFOR;
    DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMAXSB __m512i _mm512_max_epi8( __m512i a, __m512i b);
VPMAXSB __m512i _mm512_mask_max_epi8( __m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSB __m512i _mm512_maskz_max_epi8( __mmask64 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_max_epi16( __m512i a, __m512i b);
VPMAXSW __m512i _mm512_mask_max_epi16( __m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi16( __mmask32 k, __m512i a, __m512i b);
VPMAXSB __m256i _mm256_mask_max_epi8( __m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXSB __m256i _mm256_maskz_max_epi8( __mmask32 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_mask_max_epi16( __m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_maskz_max_epi16( __mmask16 k, __m256i a, __m256i b);
VPMAXSB __m128i _mm_mask_max_epi8( __m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXSB __m128i _mm_maskz_max_epi8( __mmask16 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_mask_max_epi16( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi16( __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m256i _mm256_mask_max_epi32( __m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSD __m256i _mm256_maskz_max_epi32( __mmask16 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_mask_max_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_maskz_max_epi64( __mmask8 k, __m256i a, __m256i b);
VPMAXSD __m128i _mm_mask_max_epi32( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m128i _mm_maskz_max_epi32( __mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_mask_max_epi64( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_maskz_max_epi64( __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m512i _mm512_max_epi32( __m512i a, __m512i b);
VPMAXSD __m512i _mm512_mask_max_epi32( __m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXSD __m512i _mm512_maskz_max_epi32( __mmask16 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_max_epi64( __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_mask_max_epi64( __m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_maskz_max_epi64( __mmask8 k, __m512i a, __m512i b);
(V)VPMAXSB __m128i _mm_max_epi8 ( __m128i a, __m128i b);
(V)VPMAXSW __m128i _mm_max_epi16 ( __m128i a, __m128i b);
(V)VPMAXSD __m128i _mm_max_epi32 ( __m128i a, __m128i b);
VPMAXSB __m256i _mm256_max_epi8 ( __m256i a, __m256i b);
VPMAXSW __m256i _mm256_max_epi16 ( __m256i a, __m256i b);
VPMAXSD __m256i _mm256_max_epi32 ( __m256i a, __m256i b);
PMAWSW: __m64 _mm_max_pi16( __m64 a, __m64 b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPMAXSD/Q, see Exceptions Type E4.

EVEX-encoded VPMAXSB/W, see Exceptions Type E4.nb.

## PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DE /r <sup>1</sup> PMAXUB <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSE	Compare unsigned byte integers in <i>mm2/m64</i> and <i>mm1</i> and returns maximum values.
66 0F DE /r PMAXUB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Compare packed unsigned byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed maximum values in <i>xmm1</i> .
66 0F 38 3E/r PMAXUW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE4_1	Compare packed unsigned word integers in <i>xmm2/m128</i> and <i>xmm1</i> and stores maximum packed values in <i>xmm1</i> .
VEX.NDS.128.66.0F DE /r VPMAXUB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed unsigned byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38 3E/r VPMAXUW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed unsigned word integers in <i>xmm3/m128</i> and <i>xmm2</i> and store maximum packed values in <i>xmm1</i> .
VEX.NDS.256.66.0F DE /r VPMAXUB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Compare packed unsigned byte integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed maximum values in <i>ymm1</i> .
VEX.NDS.256.66.0F38 3E/r VPMAXUW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Compare packed unsigned word integers in <i>ymm3/m256</i> and <i>ymm2</i> and store maximum packed values in <i>ymm1</i> .
EVEX.NDS.128.66.0F.WIG DE /r VPMAXUB <i>xmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG DE /r VPMAXUB <i>ymm1</i> { <i>k1</i> }[ <i>z</i> ], <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed maximum values in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG DE /r VPMAXUB <i>zmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Compare packed unsigned byte integers in <i>zmm2</i> and <i>zmm3/m512</i> and store packed maximum values in <i>zmm1</i> under writemask <i>k1</i> .
EVEX.NDS.128.66.0F38.WIG 3E /r VPMAXUW <i>xmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed maximum values in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F38.WIG 3E /r VPMAXUW <i>ymm1</i> { <i>k1</i> }[ <i>z</i> ], <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed maximum values in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F38.WIG 3E /r VPMAXUW <i>zmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Compare packed unsigned word integers in <i>zmm2</i> and <i>zmm3/m512</i> and store packed maximum values in <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA



**Description**

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/ymm/XMM register; The second source operand is a ZMM/ymm/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation****PMAXUB (64-bit operands)**

```
IF DEST[7:0] > SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] > SRC[63:56] THEN
    DEST[63:56] ← DEST[63:56];
ELSE
    DEST[63:56] ← SRC[63:56]; FI;
```

**PMAXUB (128-bit Legacy SSE version)**

```
IF DEST[7:0] > SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] > SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

**VPMAXUB (VEX.128 encoded version)**

```
IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] > SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0
```

**VPMAXUB (VEX.256 encoded version)**

```

IF SRC1[7:0] > SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] > SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:128] ← 0

```

**VPMAXUB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+7:i] ← 0
            FI
        FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**PMAXUW (128-bit Legacy SSE version)**

```

IF DEST[15:0] > SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] > SRC[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)

```

**VPMAXUW (VEX.128 encoded version)**

```

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] > SRC2[127:112] THEN
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0

```

**VPMAXUW (VEX.256 encoded version)**

```

IF SRC1[15:0] > SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] > SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:128] ← 0

```

**VPMAXUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMAXUB __m512i __mm512_max_epu8( __m512i a, __m512i b);
VPMAXUB __m512i __mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i __mm512_maskz_max_epu8( __mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i __mm512_max_epu16( __m512i a, __m512i b);
VPMAXUW __m512i __mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i __mm512_maskz_max_epu16( __mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i __mm256_mask_max_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXUB __m256i __mm256_maskz_max_epu8( __mmask32 k, __m256i a, __m256i b);
VPMAXUW __m256i __mm256_mask_max_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUW __m256i __mm256_maskz_max_epu16( __mmask16 k, __m256i a, __m256i b);
VPMAXUB __m128i __mm_mask_max_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m128i __mm_maskz_max_epu8( __mmask16 k, __m128i a, __m128i b);
VPMAXUW __m128i __mm_mask_max_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUW __m128i __mm_maskz_max_epu16( __mmask8 k, __m128i a, __m128i b);
(V)PMAXUB __m128i __mm_max_epu8 ( __m128i a, __m128i b);
(V)PMAXUW __m128i __mm_max_epu16 ( __m128i a, __m128i b);
VPMAXUB __m256i __mm256_max_epu8 ( __m256i a, __m256i b);
VPMAXUW __m256i __mm256_max_epu16 ( __m256i a, __m256i b);
PMAUXB: __m64 __mm_max_pu8(__m64 a, __m64 b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3F /r VPMAXUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.NDS.128.66.0F38.W0 3F /r VPMAXUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 3F /r VPMAXUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 3F /r VPMAXUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 3F /r VPMAXUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 3F /r VPMAXUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 3F /r VPMAXUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation****PMAXUD (128-bit Legacy SSE version)**

```

IF DEST[31:0] > SRC[31:0] THEN
    DEST[31:0] ← DEST[31:0];
ELSE
    DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:96] > SRC[127:96] THEN
    DEST[127:96] ← DEST[127:96];
ELSE
    DEST[127:96] ← SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)

```

**VPMAXUD (VEX.128 encoded version)**

```

IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:96] > SRC2[127:96] THEN
    DEST[127:96] ← SRC1[127:96];
ELSE
    DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAXVL-1:128] ← 0

```

**VPMAXUD (VEX.256 encoded version)**

```

IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] > SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0

```

**VPMAXUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[i+31:i] &gt; SRC2[31:0]

THEN DEST[i+31:i] ← SRC1[i+31:i];

ELSE DEST[i+31:i] ← SRC2[31:0];

FI;

ELSE

IF SRC1[i+31:i] &gt; SRC2[i+31:i]

THEN DEST[i+31:i] ← SRC1[i+31:i];

ELSE DEST[i+31:i] ← SRC2[i+31:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[i+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**VPMAXUQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[i+63:i] &gt; SRC2[63:0]

THEN DEST[i+63:i] ← SRC1[i+63:i];

ELSE DEST[i+63:i] ← SRC2[63:0];

FI;

ELSE

IF SRC1[i+31:i] &gt; SRC2[i+31:i]

THEN DEST[i+63:i] ← SRC1[i+63:i];

ELSE DEST[i+63:i] ← SRC2[i+63:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[i+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMAXUD __m512i _mm512_max_epu32(__m512i a, __m512i b);
VPMAXUD __m512i _mm512_mask_max_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXUD __m512i _mm512_maskz_max_epu32(__mmask16 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_max_epu64(__m512i a, __m512i b);
VPMAXUQ __m512i _mm512_mask_max_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_maskz_max_epu64(__mmask8 k, __m512i a, __m512i b);
VPMAXUD __m256i _mm256_mask_max_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUD __m256i _mm256_maskz_max_epu32(__mmask16 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_mask_max_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_maskz_max_epu64(__mmask8 k, __m256i a, __m256i b);
VPMAXUD __m128i _mm_mask_max_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUD __m128i _mm_maskz_max_epu32(__mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_mask_max_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUD __m128i _mm_max_epu32 (__m128i a, __m128i b);
VPMAXUD __m256i _mm256_max_epu32 (__m256i a, __m256i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



## PMINSB/PMINSW—Minimum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EA /r <sup>1</sup> PMINSW mm1, mm2/m64	A	V/V	SSE	Compare signed word integers in mm2/m64 and mm1 and return minimum values.
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38 38 /r VPMINSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F EA /r VPMINSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.256.66.0F38 38 /r VPMINSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.NDS.256.66.0F EA /r VPMINSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.WIG 38 /r VPMINSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 38 /r VPMINSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 38 /r VPMINSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG EA /r VPMINSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EA /r VPMINSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG EA /r VPMINSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
<b>NOTES:</b> 1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A</i> and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A</i> .				

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version **PMINSW**: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

## Operation

### PMINSW (64-bit operands)

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
IF DEST[63:48] < SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; FI;
```

### PMINSB (128-bit Legacy SSE version)

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

### VPMINSB (VEX.128 encoded version)

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0
```

**VPMINSB (VEX.256 encoded version)**

```

IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] < SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:256] ← 0

```

**VPMINSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**PMINSW (128-bit Legacy SSE version)**

```

IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] < SRC[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)

```

**VPMINSW (VEX.128 encoded version)**

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] < SRC2[127:112] THEN
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0

```

**VPMINSW (VEX.256 encoded version)**

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] < SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:256] ← 0

```

**VPMINSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMINSB __m512i __mm512_min_epi8( __m512i a, __m512i b);
VPMINSB __m512i __mm512_mask_min_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINSB __m512i __mm512_maskz_min_epi8( __mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i __mm512_min_epi16( __m512i a, __m512i b);
VPMINSW __m512i __mm512_mask_min_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINSW __m512i __mm512_maskz_min_epi16( __mmask32 k, __m512i a, __m512i b);
VPMINSB __m256i __mm256_mask_min_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINSB __m256i __mm256_maskz_min_epi8( __mmask32 k, __m256i a, __m256i b);
VPMINSW __m256i __mm256_mask_min_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSW __m256i __mm256_maskz_min_epi16( __mmask16 k, __m256i a, __m256i b);
VPMINSB __m128i __mm_mask_min_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINSB __m128i __mm_maskz_min_epi8( __mmask16 k, __m128i a, __m128i b);
VPMINSW __m128i __mm_mask_min_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSW __m128i __mm_maskz_min_epi16( __mmask8 k, __m128i a, __m128i b);
(V)PMINSB __m128i __mm_min_epi8 ( __m128i a, __m128i b);
(V)PMINSW __m128i __mm_min_epi16 ( __m128i a, __m128i b);
VPMINSB __m256i __mm256_min_epi8 ( __m256i a, __m256i b);
VPMINSW __m256i __mm256_min_epi16 ( __m256i a, __m256i b);
PMINSW: __m64 __mm_min_pi16 ( __m64 a, __m64 b)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

## PMINSD/PMINSQ—Minimum of Packed Signed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.W0 39 /r VPMINSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 39 /r VPMINSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 39 /r VPMINSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 39 /r VPMINSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed signed qword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 39 /r VPMINSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed signed qword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 39 /r VPMINSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed signed dword or qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

### Operation

#### PMINSD (128-bit Legacy SSE version)

```
IF DEST[31:0] < SRC[31:0] THEN
    DEST[31:0] ← DEST[31:0];
ELSE
    DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:96] < SRC[127:96] THEN
    DEST[127:96] ← DEST[127:96];
ELSE
    DEST[127:96] ← SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

#### VPMINSD (VEX.128 encoded version)

```
IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:96] < SRC2[127:96] THEN
    DEST[127:96] ← SRC1[127:96];
ELSE
    DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAXVL-1:128] ← 0
```

#### VPMINSD (VEX.256 encoded version)

```
IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] < SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0
```

**VPMINSD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[j+31:i] &lt; SRC2[31:0]

THEN DEST[j+31:i] ← SRC1[j+31:i];

ELSE DEST[j+31:i] ← SRC2[31:0];

FI;

ELSE

IF SRC1[j+31:i] &lt; SRC2[i+31:i]

THEN DEST[j+31:i] ← SRC1[j+31:i];

ELSE DEST[j+31:i] ← SRC2[i+31:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**VPMINSQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[j+63:i] &lt; SRC2[63:0]

THEN DEST[j+63:i] ← SRC1[j+63:i];

ELSE DEST[j+63:i] ← SRC2[63:0];

FI;

ELSE

IF SRC1[j+63:i] &lt; SRC2[i+63:i]

THEN DEST[j+63:i] ← SRC1[j+63:i];

ELSE DEST[j+63:i] ← SRC2[i+63:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMINSQ __m512i _mm512_min_epi32(__m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi32(__mmask16 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_min_epi64(__m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSQ __m256i _mm256_mask_min_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_maskz_min_epi32(__mmask16 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_mask_min_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_maskz_min_epi64(__mmask8 k, __m256i a, __m256i b);
VPMINSQ __m128i _mm_mask_min_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_maskz_min_epi32(__mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_mask_min_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_maskz_min_epi64(__mmask8 k, __m128i a, __m128i b);
(V)PMINSQ __m128i _mm_min_epi32(__m128i a, __m128i b);
VPMINSQ __m256i _mm256_min_epi32(__m256i a, __m256i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMINUB/PMINUW—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DA /r <sup>1</sup> PMINUB <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSE	Compare unsigned byte integers in <i>mm2/m64</i> and <i>mm1</i> and returns minimum values.
66 0F DA /r PMINUB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Compare packed unsigned byte integers in <i>xmm1</i> and <i>xmm2/m128</i> and store packed minimum values in <i>xmm1</i> .
66 0F 38 3A/r PMINUW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE4_1	Compare packed unsigned word integers in <i>xmm2/m128</i> and <i>xmm1</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F DA /r VPMINUB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed unsigned byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed minimum values in <i>xmm1</i> .
VEX.NDS.128.66.0F38 3A/r VPMINUW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Compare packed unsigned word integers in <i>xmm3/m128</i> and <i>xmm2</i> and return packed minimum values in <i>xmm1</i> .
VEX.NDS.256.66.0F DA /r VPMINUB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Compare packed unsigned byte integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed minimum values in <i>ymm1</i> .
VEX.NDS.256.66.0F38 3A/r VPMINUW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Compare packed unsigned word integers in <i>ymm3/m256</i> and <i>ymm2</i> and return packed minimum values in <i>ymm1</i> .
EVEX.NDS.128.66.0F DA /r VPMINUB <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in <i>xmm2</i> and <i>xmm3/m128</i> and store packed minimum values in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F DA /r VPMINUB <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in <i>ymm2</i> and <i>ymm3/m256</i> and store packed minimum values in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F DA /r VPMINUB <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Compare packed unsigned byte integers in <i>zmm2</i> and <i>zmm3/m512</i> and store packed minimum values in <i>zmm1</i> under writemask <i>k1</i> .
EVEX.NDS.128.66.0F38 3A/r VPMINUW <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in <i>xmm3/m128</i> and <i>xmm2</i> and return packed minimum values in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F38 3A/r VPMINUW <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in <i>ymm3/m256</i> and <i>ymm2</i> and return packed minimum values in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F38 3A/r VPMINUW <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Compare packed unsigned word integers in <i>zmm3/m512</i> and <i>zmm2</i> and return packed minimum values in <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
B	NA	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
C	Full Mem	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

**Description**

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version **PMINUB**: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation****PMINUB (for 64-bit operands)**

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] < SRC[63:56] THEN
    DEST[63:56] ← DEST[63:56];
ELSE
    DEST[63:56] ← SRC[63:56]; FI;
```

**PMINUB instruction for 128-bit operands:**

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

**VPMINUB (VEX.128 encoded version)**

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0
```

**VPMINUB (VEX.256 encoded version)**

```

IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[15:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] < SRC2[255:248] THEN
    DEST[255:248] ← SRC1[255:248];
ELSE
    DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:256] ← 0

```

**VPMINUB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+7:i] ← 0
            FI
        FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**PMINUW instruction for 128-bit operands:**

```

IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:112] < SRC[127:112] THEN
    DEST[127:112] ← DEST[127:112];
ELSE
    DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)

```

**VPMINUW (VEX.128 encoded version)**

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF SRC1[127:112] < SRC2[127:112] THEN
    DEST[127:112] ← SRC1[127:112];
ELSE
    DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0

```

**VPMINUW (VEX.256 encoded version)**

```

IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] < SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:256] ← 0

```

**VPMINUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMINUB __m512i __mm512_min_epu8( __m512i a, __m512i b);
VPMINUB __m512i __mm512_mask_min_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINUB __m512i __mm512_maskz_min_epu8( __mmask64 k, __m512i a, __m512i b);
VPMINUW __m512i __mm512_min_epu16( __m512i a, __m512i b);
VPMINUW __m512i __mm512_mask_min_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINUW __m512i __mm512_maskz_min_epu16( __mmask32 k, __m512i a, __m512i b);
VPMINUB __m256i __mm256_mask_min_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINUB __m256i __mm256_maskz_min_epu8( __mmask32 k, __m256i a, __m256i b);
VPMINUW __m256i __mm256_mask_min_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUW __m256i __mm256_maskz_min_epu16( __mmask16 k, __m256i a, __m256i b);
VPMINUB __m128i __mm_mask_min_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINUB __m128i __mm_maskz_min_epu8( __mmask16 k, __m128i a, __m128i b);
VPMINUW __m128i __mm_mask_min_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUW __m128i __mm_maskz_min_epu16( __mmask8 k, __m128i a, __m128i b);
(V)PMINUB __m128i __mm_min_epu8 ( __m128i a, __m128i b)
(V)PMINUW __m128i __mm_min_epu16 ( __m128i a, __m128i b);
VPMINUB __m256i __mm256_min_epu8 ( __m256i a, __m256i b)
VPMINUW __m256i __mm256_min_epu16 ( __m256i a, __m256i b);
PMINUB: __m64 __m_min_pu8 (__m64 a, __m64 b)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

**PMINUD/PMINUQ—Minimum of Packed Unsigned Integers**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3B /r VPMINUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.W0 3B /r VPMINUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 3B /r VPMINUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 3B /r VPMINUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 3B /r VPMINUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 3B /r VPMINUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 3B /r VPMINUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

**Operation****PMINUD (128-bit Legacy SSE version)**

PMINUD instruction for 128-bit operands:

IF DEST[31:0] < SRC[31:0] THEN

DEST[31:0] ← DEST[31:0];

ELSE

DEST[31:0] ← SRC[31:0]; FI;

(\* Repeat operation for 2nd through 7th words in source and destination operands \*)

IF DEST[127:96] < SRC[127:96] THEN

DEST[127:96] ← DEST[127:96];

ELSE

DEST[127:96] ← SRC[127:96]; FI;

DEST[MAXVL-1:128] (Unmodified)

**VPMINUD (VEX.128 encoded version)**

VPMINUD instruction for 128-bit operands:

IF SRC1[31:0] < SRC2[31:0] THEN

DEST[31:0] ← SRC1[31:0];

ELSE

DEST[31:0] ← SRC2[31:0]; FI;

(\* Repeat operation for 2nd through 3rd dwords in source and destination operands \*)

IF SRC1[127:96] < SRC2[127:96] THEN

DEST[127:96] ← SRC1[127:96];

ELSE

DEST[127:96] ← SRC2[127:96]; FI;

DEST[MAXVL-1:128] ← 0

**VPMINUD (VEX.256 encoded version)**

VPMINUD instruction for 128-bit operands:

IF SRC1[31:0] < SRC2[31:0] THEN

DEST[31:0] ← SRC1[31:0];

ELSE

DEST[31:0] ← SRC2[31:0]; FI;

(\* Repeat operation for 2nd through 7th dwords in source and destination operands \*)

IF SRC1[255:224] < SRC2[255:224] THEN

DEST[255:224] ← SRC1[255:224];

ELSE

DEST[255:224] ← SRC2[255:224]; FI;

DEST[MAXVL-1:256] ← 0



**VPMINUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[i+31:i] &lt; SRC2[31:0]

THEN DEST[i+31:i] ← SRC1[i+31:i];

ELSE DEST[i+31:i] ← SRC2[31:0];

FI;

ELSE

IF SRC1[i+31:i] &lt; SRC2[i+31:i]

THEN DEST[i+31:i] ← SRC1[i+31:i];

ELSE DEST[i+31:i] ← SRC2[i+31:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**VPMINUQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

IF SRC1[i+63:i] &lt; SRC2[63:0]

THEN DEST[i+63:i] ← SRC1[i+63:i];

ELSE DEST[i+63:i] ← SRC2[63:0];

FI;

ELSE

IF SRC1[i+63:i] &lt; SRC2[i+63:i]

THEN DEST[i+63:i] ← SRC1[i+63:i];

ELSE DEST[i+63:i] ← SRC2[i+63:i];

FI;

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMINUD __m512i _mm512_min_epu32( __m512i a, __m512i b);
VPMINUD __m512i _mm512_mask_min_epu32( __m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINUD __m512i _mm512_maskz_min_epu32( __mmask16 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_min_epu64( __m512i a, __m512i b);
VPMINUQ __m512i _mm512_mask_min_epu64( __m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_maskz_min_epu64( __mmask8 k, __m512i a, __m512i b);
VPMINUD __m256i _mm256_mask_min_epu32( __m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUD __m256i _mm256_maskz_min_epu32( __mmask16 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_mask_min_epu64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_maskz_min_epu64( __mmask8 k, __m256i a, __m256i b);
VPMINUD __m128i _mm_mask_min_epu32( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUD __m128i _mm_maskz_min_epu32( __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_mask_min_epu64( __m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_maskz_min_epu64( __mmask8 k, __m128i a, __m128i b);
(V)PMINUD __m128i _mm_min_epu32 ( __m128i a, __m128i b);
VPMINUD __m256i _mm256_min_epu32 ( __m256i a, __m256i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMOVMASKB—Move Byte Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D7 /r <sup>1</sup> PMOVMASKB reg, mm	RM	V/V	SSE	Move a byte mask of mm to reg. The upper bits of r32 or r64 are zeroed
66 0F D7 /r PMOVMASKB reg, xmm	RM	V/V	SSE2	Move a byte mask of xmm to reg. The upper bits of r32 or r64 are zeroed
VEX.128.66.0F.WIG D7 /r VPMOVMASKB reg, xmm1	RM	V/V	AVX	Move a byte mask of xmm1 to reg. The upper bits of r32 or r64 are filled with zeros.
VEX.256.66.0F.WIG D7 /r VPMOVMASKB reg, ymm1	RM	V/V	AVX2	Move a 32-bit mask of ymm1 to reg. The upper bits of r64 are filled with zeros.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand).

The byte mask is 8 bits for 64-bit source operand, 16 bits for 128-bit source operand and 32 bits for 256-bit source operand. The destination operand is a general-purpose register.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

Legacy SSE version: The source operand is an MMX technology register.

128-bit Legacy SSE version: The source operand is an XMM register.

VEX.128 encoded version: The source operand is an XMM register.

VEX.256 encoded version: The source operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b.

### Operation

#### PMOVMASKB (with 64-bit source operand and r32)

```

r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r32[7] ← SRC[63];
r32[31:8] ← ZERO_FILL;
```

#### (V)PMOVMASKB (with 128-bit source operand and r32)

```

r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r32[15] ← SRC[127];
r32[31:16] ← ZERO_FILL;
```

**VPMOVMASKB (with 256-bit source operand and r32)**

$r32[0] \leftarrow SRC[7];$   
 $r32[1] \leftarrow SRC[15];$   
 (\* Repeat operation for bytes 3rd through 31 \*)  
 $r32[31] \leftarrow SRC[255];$

**PMOVMASKB (with 64-bit source operand and r64)**

$r64[0] \leftarrow SRC[7];$   
 $r64[1] \leftarrow SRC[15];$   
 (\* Repeat operation for bytes 2 through 6 \*)  
 $r64[7] \leftarrow SRC[63];$   
 $r64[63:8] \leftarrow ZERO\_FILL;$

**(V)PMOVMASKB (with 128-bit source operand and r64)**

$r64[0] \leftarrow SRC[7];$   
 $r64[1] \leftarrow SRC[15];$   
 (\* Repeat operation for bytes 2 through 14 \*)  
 $r64[15] \leftarrow SRC[127];$   
 $r64[63:16] \leftarrow ZERO\_FILL;$

**VPMOVMASKB (with 256-bit source operand and r64)**

$r64[0] \leftarrow SRC[7];$   
 $r64[1] \leftarrow SRC[15];$   
 (\* Repeat operation for bytes 2 through 31 \*)  
 $r64[31] \leftarrow SRC[255];$   
 $r64[63:32] \leftarrow ZERO\_FILL;$

**Intel C/C++ Compiler Intrinsic Equivalent**

PMOVMASKB:        `int _mm_movemask_pi8(__m64 a)`  
 (V)PMOVMASKB:    `int _mm_movemask_epi8 (__m128i a)`  
 VPMOVMASKB:       `int _mm256_movemask_epi8 (__m256i a)`

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

See Exceptions Type 7; additionally

#UD                      If VEX.vvvv  $\neq$  1111B.

**PMOVSX—Packed Move with Sign Extend**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 20 /r PMOVSXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 21 /r PMOVSXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 22 /r PMOVSXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 23/r PMOVSXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 24 /r PMOVSXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 25 /r PMOVSXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64	A	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1, xmm2/m32	A	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	A	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1, xmm2/m64	A	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1, xmm2/m32	A	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ xmm1, xmm2/m64	A	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1, xmm2/m32	A	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 25 /r VPMOVSXDQ ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512BW	Sign extend 8 packed 8-bit integers in xmm2/m64 to 8 packed 16-bit integers in zmm1.
EVEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 20 /r VPMOVSXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW	Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 21 /r VPMOVSXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	AVX512VL AVX512F	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	AVX512VL AVX512F	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 22 /r VPMOVSXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Sign extend 4 packed 16-bit integers in the low 8 bytes of ymm2/mem to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Sign extend 8 packed 16-bit integers in the low 16 bytes of ymm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 23 /r VPMOVSXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 24 /r VPMOVSXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 25 /r VPMOVSXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 25 /r VPMOVSXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 25 /r VPMOVSXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Quarter Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Eighth Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

## Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word or dword integers starting from the low bytes of the source operand (second operand) are sign extended to word, dword or quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

## Operation

**Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST, SRC)**

```
DEST[15:0] ← SignExtend(SRC[7:0]);
DEST[31:16] ← SignExtend(SRC[15:8]);
DEST[47:32] ← SignExtend(SRC[23:16]);
DEST[63:48] ← SignExtend(SRC[31:24]);
DEST[79:64] ← SignExtend(SRC[39:32]);
DEST[95:80] ← SignExtend(SRC[47:40]);
DEST[111:96] ← SignExtend(SRC[55:48]);
DEST[127:112] ← SignExtend(SRC[63:56]);
```

**Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST, SRC)**

```
DEST[31:0] ← SignExtend(SRC[7:0]);
DEST[63:32] ← SignExtend(SRC[15:8]);
DEST[95:64] ← SignExtend(SRC[23:16]);
DEST[127:96] ← SignExtend(SRC[31:24]);
```

**Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST, SRC)**

```
DEST[63:0] ← SignExtend(SRC[7:0]);
DEST[127:64] ← SignExtend(SRC[15:8]);
```

**Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST, SRC)**

```
DEST[31:0] ← SignExtend(SRC[15:0]);
DEST[63:32] ← SignExtend(SRC[31:16]);
DEST[95:64] ← SignExtend(SRC[47:32]);
DEST[127:96] ← SignExtend(SRC[63:48]);
```

**Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST, SRC)**

```
DEST[63:0] ← SignExtend(SRC[15:0]);
DEST[127:64] ← SignExtend(SRC[31:16]);
```

**Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST, SRC)**

```
DEST[63:0] ← SignExtend(SRC[31:0]);
DEST[127:64] ← SignExtend(SRC[63:32]);
```

**VPMOVSXBW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
```

```
IF VL >= 256
```

```
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
```

```
FI;
```

```
IF VL >= 512
```

```
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
```

```
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
```

```
FI;
```

```
FOR j ← 0 TO KL-1
```

```
    i ← j * 16
```

```
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+15:i] ← TEMP_DEST[i+15:i]
```

```
    ELSE
```

```
        IF *merging-masking* ; merging-masking
```

```
            THEN *DEST[i+15:i] remains unchanged*
```

```
            ELSE *zeroing-masking* ; zeroing-masking
```

```
                DEST[i+15:i] ← 0
```

```
    FI
```

```
FI;
```

```
ENDFOR
```

```
DEST[MAXVL-1:VL] ← 0
```

**VPMOVSXBD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
```

```
IF VL >= 256
```

```
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
```

```
FI;
```

```
IF VL >= 512
```

```
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
```

```
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
```

```
FI;
```

```
FOR j ← 0 TO KL-1
```

```
    i ← j * 32
```

```
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
```

```
    ELSE
```

```
        IF *merging-masking* ; merging-masking
```

```
            THEN *DEST[i+31:i] remains unchanged*
```

```
            ELSE *zeroing-masking* ; zeroing-masking
```

```
                DEST[i+31:i] ← 0
```

```
    FI
```

```
FI;
```

```
ENDFOR
```

```
DEST[MAXVL-1:VL] ← 0
```



**VPMOVSXBQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[127:0], SRC[15:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[255:128], SRC[31:16])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[383:256], SRC[47:32])

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(TMP\_DEST[511:384], SRC[63:48])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TEMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPMOVSXWD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[127:0], SRC[63:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[255:128], SRC[127:64])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[383:256], SRC[191:128])

Packed\_Sign\_Extend\_WORD\_to\_DWORD(TMP\_DEST[511:384], SRC[256:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TEMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPMOVSXWQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[127:0], SRC[31:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[255:128], SRC[63:32])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[383:256], SRC[95:64])

Packed\_Sign\_Extend\_WORD\_to\_QWORD(TMP\_DEST[511:384], SRC[127:96])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TEMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPMOVSXDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[127:0], SRC[63:0])

IF VL &gt;= 256

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[255:128], SRC[127:64])

FI;

IF VL &gt;= 512

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[383:256], SRC[191:128])

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(TEMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TEMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPMOVSXBW (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[127:0], SRC[63:0])

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[255:128], SRC[127:64])

DEST[MAXVL-1:256] ← 0

**VPMOVSXBD (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[31:0])  
 Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[255:128], SRC[63:32])  
 DEST[MAXVL-1:256] ← 0

**VPMOVSXBQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[15:0])  
 Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[255:128], SRC[31:16])  
 DEST[MAXVL-1:256] ← 0

**VPMOVSXWD (VEX.256 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[63:0])  
 Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[255:128], SRC[127:64])  
 DEST[MAXVL-1:256] ← 0

**VPMOVSXWQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[31:0])  
 Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[255:128], SRC[63:32])  
 DEST[MAXVL-1:256] ← 0

**VPMOVSXDQ (VEX.256 encoded version)**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[63:0])  
 Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[255:128], SRC[127:64])  
 DEST[MAXVL-1:256] ← 0

**VPMOVSXBW (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] ← 0

**VPMOVSXBD (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] ← 0

**VPMOVSXBQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] ← 0

**VPMOVSXWD (VEX.128 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] ← 0

**VPMOVSXWQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] ← 0

**VPMOVSXDQ (VEX.128 encoded version)**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] ← 0

**PMOVSXBW**

Packed\_Sign\_Extend\_BYTE\_to\_WORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] (Unmodified)

**PMOVSBXD**

Packed\_Sign\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] (Unmodified)

**PMOVSBQ**

Packed\_Sign\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] (Unmodified)

**PMOVSWD**

Packed\_Sign\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] (Unmodified)

**PMOVSWQ**

Packed\_Sign\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] (Unmodified)

**PMOVSDQ**

Packed\_Sign\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[127:0])  
 DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMOVSBW \_\_m512i \_mm512\_cvtepi8\_epi16(\_\_m512i a);  
 VPMOVSBW \_\_m512i \_mm512\_mask\_cvtepi8\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m512i b);  
 VPMOVSBW \_\_m512i \_mm512\_maskz\_cvtepi8\_epi16(\_\_mmask32 k, \_\_m512i b);  
 VPMOVSBXD \_\_m512i \_mm512\_cvtepi8\_epi32(\_\_m512i a);  
 VPMOVSBXD \_\_m512i \_mm512\_mask\_cvtepi8\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b);  
 VPMOVSBXD \_\_m512i \_mm512\_maskz\_cvtepi8\_epi32(\_\_mmask16 k, \_\_m512i b);  
 VPMOVSBQ \_\_m512i \_mm512\_cvtepi8\_epi64(\_\_m512i a);  
 VPMOVSBQ \_\_m512i \_mm512\_mask\_cvtepi8\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b);  
 VPMOVSBQ \_\_m512i \_mm512\_maskz\_cvtepi8\_epi64(\_\_mmask8 k, \_\_m512i a);  
 VPMOVSDQ \_\_m512i \_mm512\_cvtepi32\_epi64(\_\_m512i a);  
 VPMOVSDQ \_\_m512i \_mm512\_mask\_cvtepi32\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b);  
 VPMOVSDQ \_\_m512i \_mm512\_maskz\_cvtepi32\_epi64(\_\_mmask8 k, \_\_m512i a);  
 VPMOVSWD \_\_m512i \_mm512\_cvtepi16\_epi32(\_\_m512i a);  
 VPMOVSWD \_\_m512i \_mm512\_mask\_cvtepi16\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b);  
 VPMOVSWD \_\_m512i \_mm512\_maskz\_cvtepi16\_epi32(\_\_mmask16 k, \_\_m512i a);  
 VPMOVSWQ \_\_m512i \_mm512\_cvtepi16\_epi64(\_\_m512i a);  
 VPMOVSWQ \_\_m512i \_mm512\_mask\_cvtepi16\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b);  
 VPMOVSWQ \_\_m512i \_mm512\_maskz\_cvtepi16\_epi64(\_\_mmask8 k, \_\_m512i a);  
 VPMOVSBW \_\_m256i \_mm256\_cvtepi8\_epi16(\_\_m256i a);  
 VPMOVSBW \_\_m256i \_mm256\_mask\_cvtepi8\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSBW \_\_m256i \_mm256\_maskz\_cvtepi8\_epi16(\_\_mmask16 k, \_\_m256i b);  
 VPMOVSBXD \_\_m256i \_mm256\_cvtepi8\_epi32(\_\_m256i a);  
 VPMOVSBXD \_\_m256i \_mm256\_mask\_cvtepi8\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSBXD \_\_m256i \_mm256\_maskz\_cvtepi8\_epi32(\_\_mmask8 k, \_\_m256i b);  
 VPMOVSBQ \_\_m256i \_mm256\_cvtepi8\_epi64(\_\_m256i a);  
 VPMOVSBQ \_\_m256i \_mm256\_mask\_cvtepi8\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSBQ \_\_m256i \_mm256\_maskz\_cvtepi8\_epi64(\_\_mmask8 k, \_\_m256i a);  
 VPMOVSDQ \_\_m256i \_mm256\_cvtepi32\_epi64(\_\_m256i a);  
 VPMOVSDQ \_\_m256i \_mm256\_mask\_cvtepi32\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSDQ \_\_m256i \_mm256\_maskz\_cvtepi32\_epi64(\_\_mmask8 k, \_\_m256i a);  
 VPMOVSWD \_\_m256i \_mm256\_cvtepi16\_epi32(\_\_m256i a);  
 VPMOVSWD \_\_m256i \_mm256\_mask\_cvtepi16\_epi32(\_\_m256i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSWD \_\_m256i \_mm256\_maskz\_cvtepi16\_epi32(\_\_mmask16 k, \_\_m256i a);

VPMOVSXWQ \_\_m256i \_\_mm256\_cvtepi16\_epi64(\_\_m256i a);  
 VPMOVSXWQ \_\_m256i \_\_mm256\_mask\_cvtepi16\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b);  
 VPMOVSXWQ \_\_m256i \_\_mm256\_maskz\_cvtepi16\_epi64(\_\_mmask8 k, \_\_m256i a);  
 VPMOVSXBW \_\_m128i \_\_mm\_mask\_cvtepi8\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBW \_\_m128i \_\_mm\_maskz\_cvtepi8\_epi16(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBD \_\_m128i \_\_mm\_mask\_cvtepi8\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBD \_\_m128i \_\_mm\_maskz\_cvtepi8\_epi32(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBQ \_\_m128i \_\_mm\_mask\_cvtepi8\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXBQ \_\_m128i \_\_mm\_maskz\_cvtepi8\_epi64(\_\_mmask8 k, \_\_m128i a);  
 VPMOVSXDQ \_\_m128i \_\_mm\_mask\_cvtepi32\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXDQ \_\_m128i \_\_mm\_maskz\_cvtepi32\_epi64(\_\_mmask8 k, \_\_m128i a);  
 VPMOVSXWD \_\_m128i \_\_mm\_mask\_cvtepi16\_epi32(\_\_m128i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVSXWD \_\_m128i \_\_mm\_maskz\_cvtepi16\_epi32(\_\_mmask16 k, \_\_m128i a);  
 VPMOVSXWQ \_\_m128i \_\_mm\_mask\_cvtepi16\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSXWQ \_\_m128i \_\_mm\_maskz\_cvtepi16\_epi64(\_\_mmask8 k, \_\_m128i a);  
 PMOVSXBW \_\_m128i \_\_mm\_cvtepi8\_epi16 (\_\_m128i a);  
 PMOVSXBD \_\_m128i \_\_mm\_cvtepi8\_epi32 (\_\_m128i a);  
 PMOVSXBQ \_\_m128i \_\_mm\_cvtepi8\_epi64 (\_\_m128i a);  
 PMOVSXWD \_\_m128i \_\_mm\_cvtepi16\_epi32 (\_\_m128i a);  
 PMOVSXWQ \_\_m128i \_\_mm\_cvtepi16\_epi64 (\_\_m128i a);  
 PMOVSXDQ \_\_m128i \_\_mm\_cvtepi32\_epi64 (\_\_m128i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

**PMOVZX—Packed Move with Zero Extend**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 31 /r PMOVZXBW xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW xmm1, xmm2/m64	A	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 31 /r VPMOVZXBW xmm1, xmm2/m32	A	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1, xmm2/m16	A	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1, xmm2/m64	A	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1, xmm2/m32	A	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F 38.WIG 35 /r VPMOVZXDQ xmm1, xmm2/m64	A	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 31 /r VPMOVZXBW ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1, xmm2/m32	A	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 8 packed 16-bit integers xmm2/m128 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.WIG /r VPMOVBZBW xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512BW	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
EVEX.256.66.0F38.WIG 30 /r VPMOVBZBW ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 30 /r VPMOVBZBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW	Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 31 /r VPMOVBZBD xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 31 /r VPMOVBZBD ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 31 /r VPMOVBZBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 32 /r VPMOVBZBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	AVX512VL AVX512F	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 32 /r VPMOVBZBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	AVX512VL AVX512F	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 32 /r VPMOVBZBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 33 /r VPMOVBZWD xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 33 /r VPMOVBZWD ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 33 /r VPMOVBZWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Zero extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 34 /r VPMOVBZWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 34 /r VPMOVBZWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 34 /r VPMOVBZWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 35 /r VPMOVZXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 35 /r VPMOVZXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 35 /r VPMOVZXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Zero extend 8 packed 32-bit integers in ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Quarter Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Eighth Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Legacy, VEX and EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed dword integers starting from the low bytes of the source operand (second operand) are zero extended to quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### Packed\_Zero\_Extend\_BYTE\_to\_WORD(DEST, SRC)

```
DEST[15:0] ← ZeroExtend(SRC[7:0]);
DEST[31:16] ← ZeroExtend(SRC[15:8]);
DEST[47:32] ← ZeroExtend(SRC[23:16]);
DEST[63:48] ← ZeroExtend(SRC[31:24]);
DEST[79:64] ← ZeroExtend(SRC[39:32]);
DEST[95:80] ← ZeroExtend(SRC[47:40]);
DEST[111:96] ← ZeroExtend(SRC[55:48]);
DEST[127:112] ← ZeroExtend(SRC[63:56]);
```

#### Packed\_Zero\_Extend\_BYTE\_to\_DWORD(DEST, SRC)

```
DEST[31:0] ← ZeroExtend(SRC[7:0]);
DEST[63:32] ← ZeroExtend(SRC[15:8]);
DEST[95:64] ← ZeroExtend(SRC[23:16]);
DEST[127:96] ← ZeroExtend(SRC[31:24]);
```



**Packed\_Zero\_Extend\_BYTE\_to\_QWORD(DEST, SRC)**

DEST[63:0] ← ZeroExtend(SRC[7:0]);  
 DEST[127:64] ← ZeroExtend(SRC[15:8]);

**Packed\_Zero\_Extend\_WORD\_to\_DWORD(DEST, SRC)**

DEST[31:0] ← ZeroExtend(SRC[15:0]);  
 DEST[63:32] ← ZeroExtend(SRC[31:16]);  
 DEST[95:64] ← ZeroExtend(SRC[47:32]);  
 DEST[127:96] ← ZeroExtend(SRC[63:48]);

**Packed\_Zero\_Extend\_WORD\_to\_QWORD(DEST, SRC)**

DEST[63:0] ← ZeroExtend(SRC[15:0]);  
 DEST[127:64] ← ZeroExtend(SRC[31:16]);

**Packed\_Zero\_Extend\_DWORD\_to\_QWORD(DEST, SRC)**

DEST[63:0] ← ZeroExtend(SRC[31:0]);  
 DEST[127:64] ← ZeroExtend(SRC[63:32]);

**VPMOVZXBW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[127:0], SRC[63:0])

IF VL >= 256

    Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[255:128], SRC[127:64])

FI;

IF VL >= 512

    Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[383:256], SRC[191:128])

    Packed\_Zero\_Extend\_BYTE\_to\_WORD(TMP\_DEST[511:384], SRC[255:192])

FI;

FOR j ← 0 TO KL-1

    i ← j \* 16

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+15:i] ← TEMP\_DEST[i+15:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+15:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+15:i] ← 0

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPMOVZXBW (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[127:0], SRC[31:0])

IF VL >= 256

    Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[255:128], SRC[63:32])

FI;

IF VL >= 512

    Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[383:256], SRC[95:64])

    Packed\_Zero\_Extend\_BYTE\_to\_DWORD(TMP\_DEST[511:384], SRC[127:96])

FI;

FOR j ← 0 TO KL-1

    i ← j \* 32

```

IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking*          ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPMOVZXBQ (EVEX encoded versions)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPMOVZXWD (EVEX encoded versions)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking

```

```

                DEST[i+31:i] ← 0
            FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] ← 0

```

**VPMOVZXWQ (EVEX encoded versions)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPMOVZXDQ (EVEX encoded versions)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPMOVZXBW (VEX.256 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_WORD(DEST[127:0], SRC[63:0])  
 Packed\_Zero\_Extend\_BYTE\_to\_WORD(DEST[255:128], SRC[127:64])  
 DEST[MAXVL-1:256] ← 0

**VPMOVZXBW (VEX.256 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_DWORD(DEST[127:0], SRC[31:0])  
 Packed\_Zero\_Extend\_BYTE\_to\_DWORD(DEST[255:128], SRC[63:32])  
 DEST[MAXVL-1:256] ← 0

**VPMOVZXBQ (VEX.256 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_QWORD(DEST[127:0], SRC[15:0])  
 Packed\_Zero\_Extend\_BYTE\_to\_QWORD(DEST[255:128], SRC[31:16])  
 DEST[MAXVL-1:256] ← 0

**VPMOVZXWD (VEX.256 encoded version)**

Packed\_Zero\_Extend\_WORD\_to\_DWORD(DEST[127:0], SRC[63:0])  
 Packed\_Zero\_Extend\_WORD\_to\_DWORD(DEST[255:128], SRC[127:64])  
 DEST[MAXVL-1:256] ← 0

**VPMOVZXWQ (VEX.256 encoded version)**

Packed\_Zero\_Extend\_WORD\_to\_QWORD(DEST[127:0], SRC[31:0])  
 Packed\_Zero\_Extend\_WORD\_to\_QWORD(DEST[255:128], SRC[63:32])  
 DEST[MAXVL-1:256] ← 0

**VPMOVZXDQ (VEX.256 encoded version)**

Packed\_Zero\_Extend\_DWORD\_to\_QWORD(DEST[127:0], SRC[63:0])  
 Packed\_Zero\_Extend\_DWORD\_to\_QWORD(DEST[255:128], SRC[127:64])  
 DEST[MAXVL-1:256] ← 0

**VPMOVZXBW (VEX.128 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_WORD()  
 DEST[MAXVL-1:128] ← 0

**VPMOVZXBW (VEX.128 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_DWORD()  
 DEST[MAXVL-1:128] ← 0

**VPMOVZXBQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_BYTE\_to\_QWORD()  
 DEST[MAXVL-1:128] ← 0

**VPMOVZXWD (VEX.128 encoded version)**

Packed\_Zero\_Extend\_WORD\_to\_DWORD()  
 DEST[MAXVL-1:128] ← 0

**VPMOVZXWQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_WORD\_to\_QWORD()  
 DEST[MAXVL-1:128] ← 0

**VPMOVZXDQ (VEX.128 encoded version)**

Packed\_Zero\_Extend\_DWORD\_to\_QWORD()  
 DEST[MAXVL-1:128] ← 0

**PMOVZXBW**

Packed\_Zero\_Extend\_BYTE\_to\_WORD()

DEST[MAXVL-1:128] (Unmodified)

**PMOVZXBQ**

Packed\_Zero\_Extend\_BYTE\_to\_QWORD()

DEST[MAXVL-1:128] (Unmodified)

**PMOVZXBD**

Packed\_Zero\_Extend\_WORD\_to\_DWORD()

DEST[MAXVL-1:128] (Unmodified)

**PMOVZXWQ**

Packed\_Zero\_Extend\_WORD\_to\_QWORD()

DEST[MAXVL-1:128] (Unmodified)

**PMOVZXDQ**

Packed\_Zero\_Extend\_DWORD\_to\_QWORD()

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMOVZXBW __m512i __mm512_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m512i __mm512_mask_cvtepu8_epi16(__m512i a, __mmask32 k, __m256i b);
VPMOVZXBW __m512i __mm512_maskz_cvtepu8_epi16(__mmask32 k, __m256i b);
VPMOVZXBQ __m512i __mm512_cvtepu8_epi32(__m128i a);
VPMOVZXBQ __m512i __mm512_mask_cvtepu8_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXBQ __m512i __mm512_maskz_cvtepu8_epi32(__mmask16 k, __m128i b);
VPMOVZXBQ __m512i __mm512_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m512i __mm512_mask_cvtepu8_epi64(__m512i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m512i __mm512_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m512i __mm512_cvtepu32_epi64(__m256i a);
VPMOVZXDQ __m512i __mm512_mask_cvtepu32_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXDQ __m512i __mm512_maskz_cvtepu32_epi64(__mmask8 k, __m256i a);
VPMOVZXWD __m512i __mm512_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m512i __mm512_mask_cvtepu16_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXWD __m512i __mm512_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m512i __mm512_cvtepu16_epi64(__m256i a);
VPMOVZXWQ __m512i __mm512_mask_cvtepu16_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXWQ __m512i __mm512_maskz_cvtepu16_epi64(__mmask8 k, __m256i a);
VPMOVZXBW __m256i __mm256_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m256i __mm256_mask_cvtepu8_epi16(__m256i a, __mmask16 k, __m128i b);
VPMOVZXBW __m256i __mm256_maskz_cvtepu8_epi16(__mmask16 k, __m128i b);
VPMOVZXBQ __m256i __mm256_cvtepu8_epi32(__m128i a);
VPMOVZXBQ __m256i __mm256_mask_cvtepu8_epi32(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m256i __mm256_maskz_cvtepu8_epi32(__mmask8 k, __m128i b);
VPMOVZXBQ __m256i __mm256_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m256i __mm256_mask_cvtepu8_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m256i __mm256_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m256i __mm256_cvtepu32_epi64(__m128i a);
VPMOVZXDQ __m256i __mm256_mask_cvtepu32_epi64(__m256i a, __mmask8 k, __m128i b);

```

VPMOVZXDQ \_\_m256i \_\_mm256\_maskz\_cvtepu32\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXWD \_\_m256i \_\_mm256\_cvtepu16\_epi32(\_\_m128i a);  
 VPMOVZXWD \_\_m256i \_\_mm256\_mask\_cvtepu16\_epi32(\_\_m256i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXWD \_\_m256i \_\_mm256\_maskz\_cvtepu16\_epi32(\_\_mmask16 k, \_\_m128i a);  
 VPMOVZXWQ \_\_m256i \_\_mm256\_cvtepu16\_epi64(\_\_m128i a);  
 VPMOVZXWQ \_\_m256i \_\_mm256\_mask\_cvtepu16\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXWQ \_\_m256i \_\_mm256\_maskz\_cvtepu16\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXBW \_\_m128i \_\_mm\_mask\_cvtepu8\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBW \_\_m128i \_\_mm\_maskz\_cvtepu8\_epi16( \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBW \_\_m128i \_\_mm\_mask\_cvtepu8\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBW \_\_m128i \_\_mm\_maskz\_cvtepu8\_epi32( \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m128i \_\_mm\_mask\_cvtepu8\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXBQ \_\_m128i \_\_mm\_maskz\_cvtepu8\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXDQ \_\_m128i \_\_mm\_mask\_cvtepu32\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXDQ \_\_m128i \_\_mm\_maskz\_cvtepu32\_epi64( \_\_mmask8 k, \_\_m128i a);  
 VPMOVZXWD \_\_m128i \_\_mm\_mask\_cvtepu16\_epi32(\_\_m128i a, \_\_mmask16 k, \_\_m128i b);  
 VPMOVZXWD \_\_m128i \_\_mm\_maskz\_cvtepu16\_epi32(\_\_mmask8 k, \_\_m128i a);  
 VPMOVZXWQ \_\_m128i \_\_mm\_mask\_cvtepu16\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVZXWQ \_\_m128i \_\_mm\_maskz\_cvtepu16\_epi64( \_\_mmask8 k, \_\_m128i a);  
 PMOVZXBW \_\_m128i \_\_mm\_ cvtepu8\_epi16 ( \_\_m128i a);  
 PMOVZXBW \_\_m128i \_\_mm\_ cvtepu8\_epi32 ( \_\_m128i a);  
 PMOVZXBQ \_\_m128i \_\_mm\_ cvtepu8\_epi64 ( \_\_m128i a);  
 PMOVZXWD \_\_m128i \_\_mm\_ cvtepu16\_epi32 ( \_\_m128i a);  
 PMOVZXWQ \_\_m128i \_\_mm\_ cvtepu16\_epi64 ( \_\_m128i a);  
 PMOVZXDQ \_\_m128i \_\_mm\_ cvtepu32\_epi64 ( \_\_m128i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

## PMULDQ—Multiply Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.NDS.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.NDS.128.66.0F38.W1 28 /r VPMULDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 28 /r VPMULDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 28 /r VPMULDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

### Operation

#### VPMULDQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN

            IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

                THEN DEST[i+63:i] ← SignExtend64( SRC1[i+31:i]) \* SignExtend64( SRC2[31:0])

                ELSE DEST[i+63:i] ← SignExtend64( SRC1[i+31:i]) \* SignExtend64( SRC2[i+31:i])

            FI;

        ELSE

            IF \*merging-masking\* ; merging-masking

                THEN \*DEST[i+63:i] remains unchanged\*

                ELSE \*zeroing-masking\* ; zeroing-masking

                    DEST[i+63:i] ← 0

            FI

    FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

#### VPMULDQ (VEX.256 encoded version)

DEST[63:0] ← SignExtend64( SRC1[31:0]) \* SignExtend64( SRC2[31:0])

DEST[127:64] ← SignExtend64( SRC1[95:64]) \* SignExtend64( SRC2[95:64])

DEST[191:128] ← SignExtend64( SRC1[159:128]) \* SignExtend64( SRC2[159:128])

DEST[255:192] ← SignExtend64( SRC1[223:192]) \* SignExtend64( SRC2[223:192])

DEST[MAXVL-1:256] ← 0

#### VPMULDQ (VEX.128 encoded version)

DEST[63:0] ← SignExtend64( SRC1[31:0]) \* SignExtend64( SRC2[31:0])

DEST[127:64] ← SignExtend64( SRC1[95:64]) \* SignExtend64( SRC2[95:64])

DEST[MAXVL-1:128] ← 0

#### PMULDQ (128-bit Legacy SSE version)

DEST[63:0] ← SignExtend64( DEST[31:0]) \* SignExtend64( SRC[31:0])

DEST[127:64] ← SignExtend64( SRC[95:64]) \* SignExtend64( SRC[95:64])

DEST[MAXVL-1:128] (Unmodified)

### Intel C/C++ Compiler Intrinsic Equivalent

VPMULDQ \_\_m512i \_\_mm512\_mul\_epi32(\_\_m512i a, \_\_m512i b);

VPMULDQ \_\_m512i \_\_mm512\_mask\_mul\_epi32(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPMULDQ \_\_m512i \_\_mm512\_maskz\_mul\_epi32(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPMULDQ \_\_m256i \_\_mm256\_mask\_mul\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPMULDQ \_\_m256i \_\_mm256\_mask\_mul\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPMULDQ \_\_m128i \_\_mm\_mask\_mul\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMULDQ \_\_m128i \_\_mm\_mask\_mul\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

(V)PMULDQ \_\_m128i \_\_mm\_mul\_epi32(\_\_m128i a, \_\_m128i b);

VPMULDQ \_\_m256i \_\_mm256\_mul\_epi32(\_\_m256i a, \_\_m256i b);



### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMULHRSW — Packed Multiply High with Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 38 0B /r <sup>1</sup> PMULHRSW <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>mm1</i> .
66 0F 38 0B /r PMULHRSW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>xmm1</i> .
VEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>xmm1</i> .
VEX.NDS.256.66.0F38.WIG 0B /r VPMULHRSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>ymm1</i> .
EVEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.0F38.WIG 0B /r VPMULHRSW <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.0F38.WIG 0B /r VPMULHRSW <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
B	NA	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
C	Full Mem	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

### Description

PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

## Operation

### PMULHRSW (with 64-bit operands)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
```

### PMULHRSW (with 128-bit operand)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112]) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
DEST[79:64] = temp4[16:1];
DEST[95:80] = temp5[16:1];
DEST[111:96] = temp6[16:1];
DEST[127:112] = temp7[16:1];
```

### VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112]) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
```

$\text{DEST}[63:48] \leftarrow \text{temp3}[16:1]$   
 $\text{DEST}[79:64] \leftarrow \text{temp4}[16:1]$   
 $\text{DEST}[95:80] \leftarrow \text{temp5}[16:1]$   
 $\text{DEST}[111:96] \leftarrow \text{temp6}[16:1]$   
 $\text{DEST}[127:112] \leftarrow \text{temp7}[16:1]$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$

#### VPMULHRWSW (VEX.256 encoded version)

$\text{temp0}[31:0] \leftarrow \text{INT32}((\text{SRC1}[15:0] * \text{SRC2}[15:0]) \gg 14) + 1$   
 $\text{temp1}[31:0] \leftarrow \text{INT32}((\text{SRC1}[31:16] * \text{SRC2}[31:16]) \gg 14) + 1$   
 $\text{temp2}[31:0] \leftarrow \text{INT32}((\text{SRC1}[47:32] * \text{SRC2}[47:32]) \gg 14) + 1$   
 $\text{temp3}[31:0] \leftarrow \text{INT32}((\text{SRC1}[63:48] * \text{SRC2}[63:48]) \gg 14) + 1$   
 $\text{temp4}[31:0] \leftarrow \text{INT32}((\text{SRC1}[79:64] * \text{SRC2}[79:64]) \gg 14) + 1$   
 $\text{temp5}[31:0] \leftarrow \text{INT32}((\text{SRC1}[95:80] * \text{SRC2}[95:80]) \gg 14) + 1$   
 $\text{temp6}[31:0] \leftarrow \text{INT32}((\text{SRC1}[111:96] * \text{SRC2}[111:96]) \gg 14) + 1$   
 $\text{temp7}[31:0] \leftarrow \text{INT32}((\text{SRC1}[127:112] * \text{SRC2}[127:112]) \gg 14) + 1$   
 $\text{temp8}[31:0] \leftarrow \text{INT32}((\text{SRC1}[143:128] * \text{SRC2}[143:128]) \gg 14) + 1$   
 $\text{temp9}[31:0] \leftarrow \text{INT32}((\text{SRC1}[159:144] * \text{SRC2}[159:144]) \gg 14) + 1$   
 $\text{temp10}[31:0] \leftarrow \text{INT32}((\text{SRC1}[175:160] * \text{SRC2}[175:160]) \gg 14) + 1$   
 $\text{temp11}[31:0] \leftarrow \text{INT32}((\text{SRC1}[191:176] * \text{SRC2}[191:176]) \gg 14) + 1$   
 $\text{temp12}[31:0] \leftarrow \text{INT32}((\text{SRC1}[207:192] * \text{SRC2}[207:192]) \gg 14) + 1$   
 $\text{temp13}[31:0] \leftarrow \text{INT32}((\text{SRC1}[223:208] * \text{SRC2}[223:208]) \gg 14) + 1$   
 $\text{temp14}[31:0] \leftarrow \text{INT32}((\text{SRC1}[239:224] * \text{SRC2}[239:224]) \gg 14) + 1$   
 $\text{temp15}[31:0] \leftarrow \text{INT32}((\text{SRC1}[255:240] * \text{SRC2}[255:240]) \gg 14) + 1$

$\text{DEST}[15:0] \leftarrow \text{temp0}[16:1]$   
 $\text{DEST}[31:16] \leftarrow \text{temp1}[16:1]$   
 $\text{DEST}[47:32] \leftarrow \text{temp2}[16:1]$   
 $\text{DEST}[63:48] \leftarrow \text{temp3}[16:1]$   
 $\text{DEST}[79:64] \leftarrow \text{temp4}[16:1]$   
 $\text{DEST}[95:80] \leftarrow \text{temp5}[16:1]$   
 $\text{DEST}[111:96] \leftarrow \text{temp6}[16:1]$   
 $\text{DEST}[127:112] \leftarrow \text{temp7}[16:1]$   
 $\text{DEST}[143:128] \leftarrow \text{temp8}[16:1]$   
 $\text{DEST}[159:144] \leftarrow \text{temp9}[16:1]$   
 $\text{DEST}[175:160] \leftarrow \text{temp10}[16:1]$   
 $\text{DEST}[191:176] \leftarrow \text{temp11}[16:1]$   
 $\text{DEST}[207:192] \leftarrow \text{temp12}[16:1]$   
 $\text{DEST}[223:208] \leftarrow \text{temp13}[16:1]$   
 $\text{DEST}[239:224] \leftarrow \text{temp14}[16:1]$   
 $\text{DEST}[255:240] \leftarrow \text{temp15}[16:1]$   
 $\text{DEST}[\text{MAXVL}-1:256] \leftarrow 0$

#### VPMULHRWSW (EVEX encoded version)

$(\text{KL}, \text{VL}) = (8, 128), (16, 256), (32, 512)$

FOR  $j \leftarrow 0$  TO  $\text{KL}-1$

$i \leftarrow j * 16$

IF  $k1[j]$  OR \*no writemask\*

THEN

$\text{temp}[31:0] \leftarrow ((\text{SRC1}[i+15:i] * \text{SRC2}[i+15:i]) \gg 14) + 1$

$\text{DEST}[i+15:i] \leftarrow \text{tmp}[16:1]$

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

```

        ELSE *zeroing-masking*          ; zeroing-masking
            DEST[j+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPMULHRSW __m512i _mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i _mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i _mm512_maskz_mulhrs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i _mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i _mm256_maskz_mulhrs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i _mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHRSW __m128i _mm_maskz_mulhrs_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHRSW: __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
(V)PMULHRSW: __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
VPMULHRSW: __m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)

```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF E4 /r <sup>1</sup> PMULHUW mm1, mm2/m64	A	V/V	SSE	Multiply the packed unsigned word integers in mm1 register and mm2/m64, and store the high 16 bits of the results in mm1.
66 OF E4 /r PMULHUW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed unsigned word integers in xmm1 and xmm2/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.128.66.OF.WIG E4 /r VPMULHUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1.
VEX.NDS.256.66.OF.WIG E4 /r VPMULHUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1.
EVEX.NDS.128.66.OF.WIG E4 /r VPMULHUW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.NDS.256.66.OF.WIG E4 /r VPMULHUW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.NDS.512.66.OF.WIG E4 /r VPMULHUW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed unsigned word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/ymm/xmm register. The second source operand can be a ZMM/ymm/xmm register, a 512/256/128-bit memory location. The destination operand is a ZMM/ymm/xmm register conditionally updated with writemask k1.

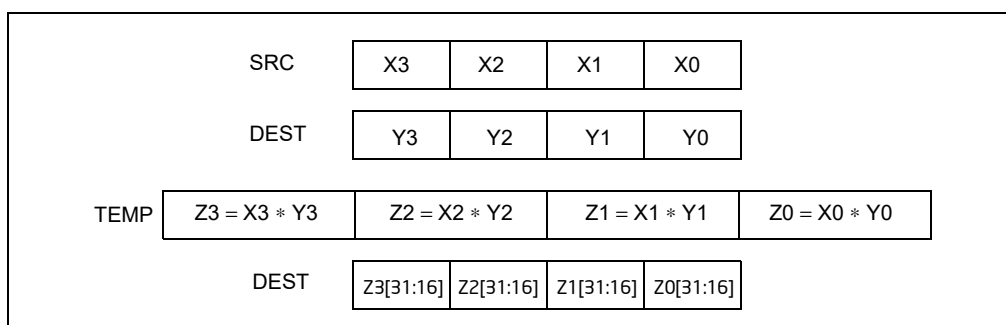


Figure 4-12. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

## Operation

### PMULHUW (with 64-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];

```

### PMULHUW (with 128-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];

```

**VPMULHUW (VEX.128 encoded version)**

TEMP0[31:0]  $\leftarrow$  SRC1[15:0] \* SRC2[15:0]  
 TEMP1[31:0]  $\leftarrow$  SRC1[31:16] \* SRC2[31:16]  
 TEMP2[31:0]  $\leftarrow$  SRC1[47:32] \* SRC2[47:32]  
 TEMP3[31:0]  $\leftarrow$  SRC1[63:48] \* SRC2[63:48]  
 TEMP4[31:0]  $\leftarrow$  SRC1[79:64] \* SRC2[79:64]  
 TEMP5[31:0]  $\leftarrow$  SRC1[95:80] \* SRC2[95:80]  
 TEMP6[31:0]  $\leftarrow$  SRC1[111:96] \* SRC2[111:96]  
 TEMP7[31:0]  $\leftarrow$  SRC1[127:112] \* SRC2[127:112]  
 DEST[15:0]  $\leftarrow$  TEMP0[31:16]  
 DEST[31:16]  $\leftarrow$  TEMP1[31:16]  
 DEST[47:32]  $\leftarrow$  TEMP2[31:16]  
 DEST[63:48]  $\leftarrow$  TEMP3[31:16]  
 DEST[79:64]  $\leftarrow$  TEMP4[31:16]  
 DEST[95:80]  $\leftarrow$  TEMP5[31:16]  
 DEST[111:96]  $\leftarrow$  TEMP6[31:16]  
 DEST[127:112]  $\leftarrow$  TEMP7[31:16]  
 DEST[MAXVL-1:128]  $\leftarrow$  0

**PMULHUW (VEX.256 encoded version)**

TEMP0[31:0]  $\leftarrow$  SRC1[15:0] \* SRC2[15:0]  
 TEMP1[31:0]  $\leftarrow$  SRC1[31:16] \* SRC2[31:16]  
 TEMP2[31:0]  $\leftarrow$  SRC1[47:32] \* SRC2[47:32]  
 TEMP3[31:0]  $\leftarrow$  SRC1[63:48] \* SRC2[63:48]  
 TEMP4[31:0]  $\leftarrow$  SRC1[79:64] \* SRC2[79:64]  
 TEMP5[31:0]  $\leftarrow$  SRC1[95:80] \* SRC2[95:80]  
 TEMP6[31:0]  $\leftarrow$  SRC1[111:96] \* SRC2[111:96]  
 TEMP7[31:0]  $\leftarrow$  SRC1[127:112] \* SRC2[127:112]  
 TEMP8[31:0]  $\leftarrow$  SRC1[143:128] \* SRC2[143:128]  
 TEMP9[31:0]  $\leftarrow$  SRC1[159:144] \* SRC2[159:144]  
 TEMP10[31:0]  $\leftarrow$  SRC1[175:160] \* SRC2[175:160]  
 TEMP11[31:0]  $\leftarrow$  SRC1[191:176] \* SRC2[191:176]  
 TEMP12[31:0]  $\leftarrow$  SRC1[207:192] \* SRC2[207:192]  
 TEMP13[31:0]  $\leftarrow$  SRC1[223:208] \* SRC2[223:208]  
 TEMP14[31:0]  $\leftarrow$  SRC1[239:224] \* SRC2[239:224]  
 TEMP15[31:0]  $\leftarrow$  SRC1[255:240] \* SRC2[255:240]  
 DEST[15:0]  $\leftarrow$  TEMP0[31:16]  
 DEST[31:16]  $\leftarrow$  TEMP1[31:16]  
 DEST[47:32]  $\leftarrow$  TEMP2[31:16]  
 DEST[63:48]  $\leftarrow$  TEMP3[31:16]  
 DEST[79:64]  $\leftarrow$  TEMP4[31:16]  
 DEST[95:80]  $\leftarrow$  TEMP5[31:16]  
 DEST[111:96]  $\leftarrow$  TEMP6[31:16]  
 DEST[127:112]  $\leftarrow$  TEMP7[31:16]  
 DEST[143:128]  $\leftarrow$  TEMP8[31:16]  
 DEST[159:144]  $\leftarrow$  TEMP9[31:16]  
 DEST[175:160]  $\leftarrow$  TEMP10[31:16]  
 DEST[191:176]  $\leftarrow$  TEMP11[31:16]  
 DEST[207:192]  $\leftarrow$  TEMP12[31:16]  
 DEST[223:208]  $\leftarrow$  TEMP13[31:16]  
 DEST[239:224]  $\leftarrow$  TEMP14[31:16]  
 DEST[255:240]  $\leftarrow$  TEMP15[31:16]  
 DEST[MAXVL-1:256]  $\leftarrow$  0



**PMULHUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN

temp[31:0] ← SRC1[j+15:i] \* SRC2[j+15:i]

DEST[j+15:i] ← tmp[31:16]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[j+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULHUW \_\_m512i \_\_mm512\_mulhi\_epu16(\_\_m512i a, \_\_m512i b);

VPMULHUW \_\_m512i \_\_mm512\_mask\_mulhi\_epu16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMULHUW \_\_m512i \_\_mm512\_maskz\_mulhi\_epu16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);

VPMULHUW \_\_m256i \_\_mm256\_mask\_mulhi\_epu16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMULHUW \_\_m256i \_\_mm256\_maskz\_mulhi\_epu16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);

VPMULHUW \_\_m128i \_\_mm\_mask\_mulhi\_epu16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMULHUW \_\_m128i \_\_mm\_maskz\_mulhi\_epu16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PMULHUW: \_\_m64 \_\_mm\_mulhi\_epu16(\_\_m64 a, \_\_m64 b)

(V)PMULHUW: \_\_m128i \_\_mm\_mulhi\_epu16 (\_\_m128i a, \_\_m128i b)

VPMULHUW: \_\_m256i \_\_mm256\_mulhi\_epu16 (\_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF E5 /r <sup>1</sup> PMULHW mm, mm/m64	A	V/V	MMX	Multiply the packed signed word integers in <i>mm1</i> register and <i>mm2/m64</i> , and store the high 16 bits of the results in <i>mm1</i> .
66 OF E5 /r PMULHW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in <i>xmm1</i> and <i>xmm2/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG E5 /r VPMULHW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> .
VEX.NDS.256.66.OF.WIG E5 /r VPMULHW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the high 16 bits of the results in <i>ymm1</i> .
EVEX.NDS.128.66.OF.WIG E5 /r VPMULHW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in <i>xmm2</i> and <i>xmm3/m128</i> , and store the high 16 bits of the results in <i>xmm1</i> under writemask <i>k1</i> .
EVEX.NDS.256.66.OF.WIG E5 /r VPMULHW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in <i>ymm2</i> and <i>ymm3/m256</i> , and store the high 16 bits of the results in <i>ymm1</i> under writemask <i>k1</i> .
EVEX.NDS.512.66.OF.WIG E5 /r VPMULHW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed signed word integers in <i>zmm2</i> and <i>zmm3/m512</i> , and store the high 16 bits of the results in <i>zmm1</i> under writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/ymm/xmm register. The second source operand can be a ZMM/ymm/xmm register, a 512/256/128-bit memory location. The destination operand is a ZMM/ymm/xmm register conditionally updated with writemask k1.

## Operation

### PMULHW (with 64-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];

```

### PMULHW (with 128-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];

```

### VPMULHW (VEX.128 encoded version)

```

TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[MAXVL-1:128] ← 0

```

**PMULHW (VEX.256 encoded version)**

TEMP0[31:0]  $\leftarrow$  SRC1[15:0] \* SRC2[15:0] (\*Signed Multiplication\*)

TEMP1[31:0]  $\leftarrow$  SRC1[31:16] \* SRC2[31:16]

TEMP2[31:0]  $\leftarrow$  SRC1[47:32] \* SRC2[47:32]

TEMP3[31:0]  $\leftarrow$  SRC1[63:48] \* SRC2[63:48]

TEMP4[31:0]  $\leftarrow$  SRC1[79:64] \* SRC2[79:64]

TEMP5[31:0]  $\leftarrow$  SRC1[95:80] \* SRC2[95:80]

TEMP6[31:0]  $\leftarrow$  SRC1[111:96] \* SRC2[111:96]

TEMP7[31:0]  $\leftarrow$  SRC1[127:112] \* SRC2[127:112]

TEMP8[31:0]  $\leftarrow$  SRC1[143:128] \* SRC2[143:128]

TEMP9[31:0]  $\leftarrow$  SRC1[159:144] \* SRC2[159:144]

TEMP10[31:0]  $\leftarrow$  SRC1[175:160] \* SRC2[175:160]

TEMP11[31:0]  $\leftarrow$  SRC1[191:176] \* SRC2[191:176]

TEMP12[31:0]  $\leftarrow$  SRC1[207:192] \* SRC2[207:192]

TEMP13[31:0]  $\leftarrow$  SRC1[223:208] \* SRC2[223:208]

TEMP14[31:0]  $\leftarrow$  SRC1[239:224] \* SRC2[239:224]

TEMP15[31:0]  $\leftarrow$  SRC1[255:240] \* SRC2[255:240]

DEST[15:0]  $\leftarrow$  TEMP0[31:16]

DEST[31:16]  $\leftarrow$  TEMP1[31:16]

DEST[47:32]  $\leftarrow$  TEMP2[31:16]

DEST[63:48]  $\leftarrow$  TEMP3[31:16]

DEST[79:64]  $\leftarrow$  TEMP4[31:16]

DEST[95:80]  $\leftarrow$  TEMP5[31:16]

DEST[111:96]  $\leftarrow$  TEMP6[31:16]

DEST[127:112]  $\leftarrow$  TEMP7[31:16]

DEST[143:128]  $\leftarrow$  TEMP8[31:16]

DEST[159:144]  $\leftarrow$  TEMP9[31:16]

DEST[175:160]  $\leftarrow$  TEMP10[31:16]

DEST[191:176]  $\leftarrow$  TEMP11[31:16]

DEST[207:192]  $\leftarrow$  TEMP12[31:16]

DEST[223:208]  $\leftarrow$  TEMP13[31:16]

DEST[239:224]  $\leftarrow$  TEMP14[31:16]

DEST[255:240]  $\leftarrow$  TEMP15[31:16]

DEST[MAXVL-1:256]  $\leftarrow$  0

**PMULHW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j  $\leftarrow$  0 TO KL-1

    i  $\leftarrow$  j \* 16

    IF k1[j] OR \*no writemask\*

        THEN

            temp[31:0]  $\leftarrow$  SRC1[i+15:i] \* SRC2[i+15:i]

            DEST[i+15:i]  $\leftarrow$  temp[31:16]

        ELSE

            IF \*merging-masking\* ; merging-masking

                THEN \*DEST[i+15:i] remains unchanged\*

                ELSE \*zeroing-masking\* ; zeroing-masking

                    DEST[i+15:i]  $\leftarrow$  0

        FI

    FI;

ENDFOR

DEST[MAXVL-1:VL]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULHW \_\_m512i \_mm512\_mulhi\_epi16(\_\_m512i a, \_\_m512i b);  
 VPMULHW \_\_m512i \_mm512\_mask\_mulhi\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULHW \_\_m512i \_mm512\_maskz\_mulhi\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPMULHW \_\_m256i \_mm256\_mask\_mulhi\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULHW \_\_m256i \_mm256\_maskz\_mulhi\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPMULHW \_\_m128i \_mm\_mask\_mulhi\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULHW \_\_m128i \_mm\_maskz\_mulhi\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PMULHW: \_\_m64 \_mm\_mulhi\_pi16(\_\_m64 m1, \_\_m64 m2)  
 (V)PMULHW: \_\_m128i \_mm\_mulhi\_epi16(\_\_m128i a, \_\_m128i b)  
 VPMULHW: \_\_m256i \_mm256\_mulhi\_epi16(\_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1.
VEX.NDS.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.NDS.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1.
EVEX.NDS.128.66.0F38.W0 40 /r VPMULLD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 40 /r VPMULLD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 40 /r VPMULLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Multiply the packed dword signed integers in zmm2 and zmm3/m512/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 40 /r VPMULLQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Multiply the packed qword signed integers in zmm2 and zmm3/m512/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

### Operation

#### VPMULLQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

      THEN Temp[127:0] ← SRC1[i+63:i] \* SRC2[63:0]

      ELSE Temp[127:0] ← SRC1[i+63:i] \* SRC2[i+63:i]

    FI;

    DEST[i+63:i] ← Temp[63:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

      ELSE ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

#### VPMULLD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN Temp[63:0] ← SRC1[i+31:i] \* SRC2[31:0]

      ELSE Temp[63:0] ← SRC1[i+31:i] \* SRC2[i+31:i]

    FI;

    DEST[i+31:i] ← Temp[31:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      \*DEST[i+31:i] remains unchanged\*

      ELSE ; zeroing-masking

        DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPMULLD (VEX.256 encoded version)**

Temp0[63:0] ← SRC1[31:0] \* SRC2[31:0]  
 Temp1[63:0] ← SRC1[63:32] \* SRC2[63:32]  
 Temp2[63:0] ← SRC1[95:64] \* SRC2[95:64]  
 Temp3[63:0] ← SRC1[127:96] \* SRC2[127:96]  
 Temp4[63:0] ← SRC1[159:128] \* SRC2[159:128]  
 Temp5[63:0] ← SRC1[191:160] \* SRC2[191:160]  
 Temp6[63:0] ← SRC1[223:192] \* SRC2[223:192]  
 Temp7[63:0] ← SRC1[255:224] \* SRC2[255:224]

DEST[31:0] ← Temp0[31:0]  
 DEST[63:32] ← Temp1[31:0]  
 DEST[95:64] ← Temp2[31:0]  
 DEST[127:96] ← Temp3[31:0]  
 DEST[159:128] ← Temp4[31:0]  
 DEST[191:160] ← Temp5[31:0]  
 DEST[223:192] ← Temp6[31:0]  
 DEST[255:224] ← Temp7[31:0]  
 DEST[MAXVL-1:256] ← 0

**VPMULLD (VEX.128 encoded version)**

Temp0[63:0] ← SRC1[31:0] \* SRC2[31:0]  
 Temp1[63:0] ← SRC1[63:32] \* SRC2[63:32]  
 Temp2[63:0] ← SRC1[95:64] \* SRC2[95:64]  
 Temp3[63:0] ← SRC1[127:96] \* SRC2[127:96]  
 DEST[31:0] ← Temp0[31:0]  
 DEST[63:32] ← Temp1[31:0]  
 DEST[95:64] ← Temp2[31:0]  
 DEST[127:96] ← Temp3[31:0]  
 DEST[MAXVL-1:128] ← 0

**PMULLD (128-bit Legacy SSE version)**

Temp0[63:0] ← DEST[31:0] \* SRC[31:0]  
 Temp1[63:0] ← DEST[63:32] \* SRC[63:32]  
 Temp2[63:0] ← DEST[95:64] \* SRC[95:64]  
 Temp3[63:0] ← DEST[127:96] \* SRC[127:96]  
 DEST[31:0] ← Temp0[31:0]  
 DEST[63:32] ← Temp1[31:0]  
 DEST[95:64] ← Temp2[31:0]  
 DEST[127:96] ← Temp3[31:0]  
 DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULLD \_\_m512i \_mm512\_mullo\_epi32(\_\_m512i a, \_\_m512i b);  
 VPMULLD \_\_m512i \_mm512\_mask\_mullo\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMULLD \_\_m512i \_mm512\_maskz\_mullo\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPMULLD \_\_m256i \_mm256\_mask\_mullo\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMULLD \_\_m256i \_mm256\_maskz\_mullo\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMULLD \_\_m128i \_mm\_mask\_mullo\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULLD \_\_m128i \_mm\_maskz\_mullo\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULLD \_\_m256i \_mm256\_mullo\_epi32(\_\_m256i a, \_\_m256i b);  
 PMULLD \_\_m128i \_mm\_mullo\_epi32(\_\_m128i a, \_\_m128i b);  
 VPMULLQ \_\_m512i \_mm512\_mullo\_epi64(\_\_m512i a, \_\_m512i b);  
 VPMULLQ \_\_m512i \_mm512\_mask\_mullo\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);



VPMULLQ \_\_m512i \_mm512\_maskz\_mullo\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPMULLQ \_\_m256i \_mm256\_mullo\_epi64(\_\_m256i a, \_\_m256i b);  
 VPMULLQ \_\_m256i \_mm256\_mask\_mullo\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMULLQ \_\_m256i \_mm256\_maskz\_mullo\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPMULLQ \_\_m128i \_mm\_mullo\_epi64(\_\_m128i a, \_\_m128i b);  
 VPMULLQ \_\_m128i \_mm\_mask\_mullo\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPMULLQ \_\_m128i \_mm\_maskz\_mullo\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF D5 /r <sup>1</sup> PMULLW mm, mm/m64	A	V/V	MMX	Multiply the packed signed word integers in mm1 register and mm2/m64, and store the low 16 bits of the results in mm1.
66 OF D5 /r PMULLW xmm1, xmm2/m128	A	V/V	SSE2	Multiply the packed signed word integers in xmm1 and xmm2/m128, and store the low 16 bits of the results in xmm1.
VEX.NDS.128.66.OF.WIG D5 /r VPMULLW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.NDS.256.66.OF.WIG D5 /r VPMULLW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1.
EVEX.NDS.128.66.OF.WIG D5 /r VPMULLW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1.
EVEX.NDS.256.66.OF.WIG D5 /r VPMULLW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1.
EVEX.NDS.512.66.OF.WIG D5 /r VPMULLW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/ymm/xmm register. The second source operand is a ZMM/ymm/xmm register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

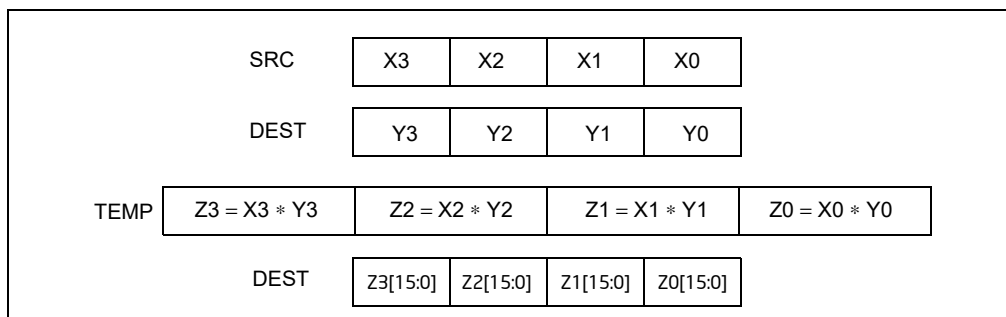


Figure 4-13. PMULLU Instruction Operation Using 64-bit Operands

## Operation

### PMULLW (with 64-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];

```

### PMULLW (with 128-bit operands)

```

TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];
DEST[79:64] ← TEMP4[15:0];
DEST[95:80] ← TEMP5[15:0];
DEST[111:96] ← TEMP6[15:0];
DEST[127:112] ← TEMP7[15:0];
DEST[MAXVL-1:256] ← 0

```

**VPMULLW (VEX.128 encoded version)**

```

Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[MAXVL-1:128] ← 0

```

**PMULLW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

    i ← j \* 16

    IF k1[j] OR \*no writemask\*

        THEN

            temp[31:0] ← SRC1[i+15:i] \* SRC2[i+15:i]

            DEST[i+15:i] ← temp[15:0]

        ELSE

            IF \*merging-masking\* ; merging-masking

                THEN \*DEST[i+15:i] remains unchanged\*

                ELSE \*zeroing-masking\* ; zeroing-masking

                    DEST[i+15:i] ← 0

        FI

    FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPMULLW __m512i _mm512_mullo_epi16(__m512i a, __m512i b);
VPMULLW __m512i _mm512_mask_mullo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULLW __m512i _mm512_maskz_mullo_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULLW __m256i _mm256_mask_mullo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULLW __m256i _mm256_maskz_mullo_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULLW __m128i _mm_mask_mullo_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLW __m128i _mm_maskz_mullo_epi16(__mmask8 k, __m128i a, __m128i b);
PMULLW: __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
(V)PMULLW: __m128i _mm_mullo_epi16 (__m128i a, __m128i b)
VPMULLW: __m256i _mm256_mullo_epi16 (__m256i a, __m256i b);

```

**Flags Affected**

None.

## **SIMD Floating-Point Exceptions**

None.

## **Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF F4 /r <sup>1</sup> PMULUDQ mm1, mm2/m64	A	V/V	SSE2	Multiply unsigned doubleword integer in mm1 by unsigned doubleword integer in mm2/m64, and store the quadword result in mm1.
66 OF F4 /r PMULUDQ xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed unsigned doubleword integers in xmm1 by packed unsigned doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.OF.WIG F4 /r VPMULUDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.NDS.256.66.OF.WIG F4 /r VPMULUDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.NDS.128.66.OF.W1 F4 /r VPMULUDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.
EVEX.NDS.256.66.OF.W1 F4 /r VPMULUDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.
EVEX.NDS.512.66.OF.W1 F4 /r VPMULUDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned

quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

## Operation

### PMULUDQ (with 64-Bit operands)

$$\text{DEST}[63:0] \leftarrow \text{DEST}[31:0] * \text{SRC}[31:0];$$

### PMULUDQ (with 128-Bit operands)

$$\begin{aligned} \text{DEST}[63:0] &\leftarrow \text{DEST}[31:0] * \text{SRC}[31:0]; \\ \text{DEST}[127:64] &\leftarrow \text{DEST}[95:64] * \text{SRC}[95:64]; \end{aligned}$$

### VPMULUDQ (VEX.128 encoded version)

$$\begin{aligned} \text{DEST}[63:0] &\leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0] \\ \text{DEST}[127:64] &\leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64] \\ \text{DEST}[\text{MAXVL}-1:128] &\leftarrow 0 \end{aligned}$$

### VPMULUDQ (VEX.256 encoded version)

$$\begin{aligned} \text{DEST}[63:0] &\leftarrow \text{SRC1}[31:0] * \text{SRC2}[31:0] \\ \text{DEST}[127:64] &\leftarrow \text{SRC1}[95:64] * \text{SRC2}[95:64] \\ \text{DEST}[191:128] &\leftarrow \text{SRC1}[159:128] * \text{SRC2}[159:128] \\ \text{DEST}[255:192] &\leftarrow \text{SRC1}[223:192] * \text{SRC2}[223:192] \\ \text{DEST}[\text{MAXVL}-1:256] &\leftarrow 0 \end{aligned}$$

**VPMULUDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i]) \* ZeroExtend64( SRC2[31:0])

ELSE DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i]) \* ZeroExtend64( SRC2[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULUDQ \_\_m512i \_\_mm512\_mul\_epu32(\_\_m512i a, \_\_m512i b);

VPMULUDQ \_\_m512i \_\_mm512\_mask\_mul\_epu32(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPMULUDQ \_\_m512i \_\_mm512\_maskz\_mul\_epu32(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);

VPMULUDQ \_\_m256i \_\_mm256\_mask\_mul\_epu32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPMULUDQ \_\_m256i \_\_mm256\_maskz\_mul\_epu32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);

VPMULUDQ \_\_m128i \_\_mm\_mask\_mul\_epu32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMULUDQ \_\_m128i \_\_mm\_maskz\_mul\_epu32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);

PMULUDQ: \_\_m64 \_\_mm\_mul\_su32(\_\_m64 a, \_\_m64 b)

(V)PMULUDQ: \_\_m128i \_\_mm\_mul\_epu32(\_\_m128i a, \_\_m128i b)

VPMULUDQ: \_\_m256i \_\_mm256\_mul\_epu32(\_\_m256i a, \_\_m256i b);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



## POP—Pop a Value from the Stack

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
8F /0	POP <i>r/m16</i>	M	Valid	Valid	Pop top of stack into <i>m16</i> ; increment stack pointer.
8F /0	POP <i>r/m32</i>	M	N.E.	Valid	Pop top of stack into <i>m32</i> ; increment stack pointer.
8F /0	POP <i>r/m64</i>	M	Valid	N.E.	Pop top of stack into <i>m64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
58+ <i>rw</i>	POP <i>r16</i>	O	Valid	Valid	Pop top of stack into <i>r16</i> ; increment stack pointer.
58+ <i>rd</i>	POP <i>r32</i>	O	N.E.	Valid	Pop top of stack into <i>r32</i> ; increment stack pointer.
58+ <i>rd</i>	POP <i>r64</i>	O	Valid	N.E.	Pop top of stack into <i>r64</i> ; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	ZO	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	ZO	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	ZO	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	ZO	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	ZO	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	ZO	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	ZO	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	ZO	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	ZO	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA
O	opcode + rd (w)	NA	NA	NA
ZO	NA	NA	NA	NA

### Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

The address size is used only when writing to a destination operand in memory.

- **Operand size.** The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).

- **Stack-address size.** Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the “Operation” section below).

A NULL value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is NULL.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0H as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt<sup>1</sup>. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). When in 64-bit mode, POPs using 32-bit operands are not encodable and POPs to DS, ES, SS are not valid. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```
IF StackAddrSize = 32
  THEN
    IF OperandSize = 32
      THEN
        DEST ← SS:ESP; (* Copy a doubleword *)
        ESP ← ESP + 4;
      ELSE (* OperandSize = 16*)
        DEST ← SS:ESP; (* Copy a word *)
```

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a POP SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that POP the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

In the following sequence, interrupts may be recognized before POP ESP executes:

```
POP SS
POP SS
POP ESP
```

```

        ESP ← ESP + 2;
    FI;
ELSE IF StackAddrSize = 64
    THEN
        IF OperandSize = 64
            THEN
                DEST ← SS:RSP; (* Copy quadword *)
                RSP ← RSP + 8;
            ELSE (* OperandSize = 16*)
                DEST ← SS:RSP; (* Copy a word *)
                RSP ← RSP + 2;
            FI;
        FI;
ELSE StackAddrSize = 16
    THEN
        IF OperandSize = 16
            THEN
                DEST ← SS:SP; (* Copy a word *)
                SP ← SP + 2;
            ELSE (* OperandSize = 32 *)
                DEST ← SS:SP; (* Copy a doubleword *)
                SP ← SP + 4;
            FI;
        FI;
FI;

```

Loading a segment register while in protected mode results in special actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

#### 64-BIT\_MODE

```

IF FS, or GS is loaded with non-NULL selector;
    THEN
        IF segment selector index is outside descriptor table limits
            OR segment is not a data or readable code segment
            OR ((segment is a data or nonconforming code segment)
                AND (both RPL and CPL > DPL))
                THEN #GP(selector);
        IF segment not marked present
            THEN #NP(selector);
    ELSE
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
    FI;
FI;
IF FS, or GS is loaded with a NULL selector;
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
FI;

```

#### PREOTECTED MODE OR COMPATIBILITY MODE;

```

IF SS is loaded;

```

```

THEN
    IF segment selector is NULL
        THEN #GP(0);
    FI;
    IF segment selector index is outside descriptor table limits
        or segment selector's RPL  $\neq$  CPL
        or segment is not a writable data segment
        or DPL  $\neq$  CPL
        THEN #GP(selector);
    FI;
    IF segment not marked present
        THEN #SS(selector);
    ELSE
        SS  $\leftarrow$  segment selector;
        SS  $\leftarrow$  segment descriptor;
    FI;
FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector;
THEN
    IF segment selector index is outside descriptor table limits
        or segment is not a data or readable code segment
        or ((segment is a data or nonconforming code segment)
            and (both RPL and CPL > DPL))
        THEN #GP(selector);
    FI;
    IF segment not marked present
        THEN #NP(selector);
    ELSE
        SegmentRegister  $\leftarrow$  segment selector;
        SegmentRegister  $\leftarrow$  segment descriptor;
    FI;
FI;

IF DS, ES, FS, or GS is loaded with a NULL selector
THEN
    SegmentRegister  $\leftarrow$  segment selector;
    SegmentRegister  $\leftarrow$  segment descriptor;
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	<p>If attempt is made to load SS register with NULL segment selector.</p> <p>If the destination operand is in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p>
#GP(selector)	<p>If segment selector index is outside descriptor table limits.</p> <p>If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.</p>

	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the current top of stack is not within the stack segment.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same as for protected mode exceptions.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#GP(selector)	If the descriptor is outside the descriptor table limit.
	If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code)	If a page fault occurs.
#NP	If the FS or GS register is being loaded and the segment pointed to is marked not present.
#UD	If the LOCK prefix is used.

## POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
61	POPA	Z0	Invalid	Valid	Pop DI, SI, BP, BX, DX, CX, and AX.
61	POPAD	Z0	Invalid	Valid	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Pops doublewords (POPAD) or words (POPA) from the stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSHA/PUSHAD instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded.

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used (using the operand-size override prefix [66H] if necessary). Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used. (The D flag in the current code segment's segment descriptor determines the operand-size attribute.)

This instruction executes as described in non-64-bit modes. It is not valid in 64-bit mode.

### Operation

IF 64-Bit Mode

THEN

#UD;

ELSE

IF OperandSize = 32 (\* Instruction = POPAD \*)

THEN

EDI ← Pop();

ESI ← Pop();

EBP ← Pop();

Increment ESP by 4; (\* Skip next 4 bytes of stack \*)

EBX ← Pop();

EDX ← Pop();

ECX ← Pop();

EAX ← Pop();

ELSE (\* OperandSize = 16, instruction = POPA \*)

DI ← Pop();

SI ← Pop();

BP ← Pop();

Increment ESP by 2; (\* Skip next 2 bytes of stack \*)

BX ← Pop();

DX ← Pop();

CX ← Pop();

AX ← Pop();

FI;

FI;

## Flags Affected

None.

## Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#SS	If the starting or ending stack address is not within the stack segment.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same as for protected mode exceptions.

## 64-Bit Mode Exceptions

#UD	If in 64-bit mode.
-----	--------------------

## POPCNT — Return the Count of Number of Bits Set to 1

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F3 0F B8 /r	POPCNT <i>r16, r/m16</i>	RM	Valid	Valid	POPCNT on <i>r/m16</i>
F3 0F B8 /r	POPCNT <i>r32, r/m32</i>	RM	Valid	Valid	POPCNT on <i>r/m32</i>
F3 REX.W 0F B8 /r	POPCNT <i>r64, r/m64</i>	RM	Valid	N.E.	POPCNT on <i>r/m64</i>

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction calculates the number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

### Operation

```
Count = 0;
For (i=0; i < OperandSize; i++)
{
    IF (SRC[ i] = 1) // i'th bit
        THEN Count++; FI;
}
DEST ← Count;
```

### Flags Affected

OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if SRC = 0, otherwise ZF is cleared.

### Intel C/C++ Compiler Intrinsic Equivalent

```
POPCNT:      int __mm_popcnt_u32(unsigned int a);
POPCNT:      int64_t __mm_popcnt_u64(unsigned __int64 a);
```

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.  
 #SS(0) If a memory operand effective address is outside the SS segment limit.  
 #PF (fault-code) For a page fault.  
 #AC(0) If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.  
 #UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.  
 If LOCK prefix is used.

### Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.  
 #SS(0) If a memory operand effective address is outside the SS segment limit.  
 #UD If CPUID.01H:ECX.POPCNT [Bit 23] = 0.  
 If LOCK prefix is used.



**Virtual 8086 Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0. If LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0. If LOCK prefix is used.

## POPF/POPFD/POPfq—Pop Stack into EFLAGS Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9D	POPF	Z0	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	Z0	N.E.	Valid	Pop top of stack into EFLAGS.
9D	POPfq	Z0	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD/PUSHfq instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16; the POPFD instruction is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size to 16 for POPF and to 32 for POPFD. Others may treat the mnemonics as synonyms (POPF/POPFD) and use the setting of the operand-size attribute to determine the size of values to pop from the stack.

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. See Table 4-15 and the key below for details.

When operating in protected, compatibility, or 64-bit mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except RF<sup>1</sup>, VIP, VIF, and VM may be modified. VIP, VIF and VM remain unaffected.

When operating in protected, compatibility, or 64-bit mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and RF, IF, VIP, VIF, and VM; these remain unaffected. The AC and ID flags can only be modified if the operand-size attribute is 32. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the POPF/POPFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), POPF (but not POPFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. POPF, however, is not affected by CR4.PVI.)

In 64-bit mode, the mnemonic assigned is POPfq (note that the 32-bit operand is not encodable). POPfq pops 64 bits from the stack. Reserved bits of RFLAGS (including the upper 32 bits of RFLAGS) are not affected.

See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS registers.

1. RF is always zero after the execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.

Table 4-15. Effect of POPF/POPFD on the EFLAGS Register

Mode	Operand Size	CPL	IOPL	Flags																Notes	
				21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2		0
				ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF		CF
Real-Address Mode (CR0.PE = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
Protected, Compatibility, and 64-Bit Modes  (CR0.PE = 1 EFLAGS.VM = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	16	1-3	<CPL	N	N	N	N	N	0	S	N	S	S	N	S	S	S	S	S	S	
	16	1-3	≥CPL	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32, 64	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32, 64	1-3	<CPL	S	N	N	S	N	0	S	N	S	S	N	S	S	S	S	S	S	
	32, 64	1-3	≥CPL	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
Virtual-8086 (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 0)	16	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
VME (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 1)	16	3	0-2	N/ X	N/ X	SV/ X	N/ X	N/ X	0/ X	S/ X	N/X	S/ X	S/ X	N/ X	S/ X	S/ X	S/ X	S/ X	S/ X	S/ X	2,3
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	

**NOTES:**

1. #GP fault - no flag update
2. #GP fault with no flag update if VIP=1 in EFLAGS register and IF=1 in FLAGS value on stack
3. #GP fault with no flag update if TF=1 in FLAGS value on stack

Key	
<b>S</b>	Updated from stack
<b>SV</b>	Updated from IF (bit 9) in FLAGS value on stack
<b>N</b>	No change in value
<b>X</b>	No EFLAGS update
<b>0</b>	Value is cleared

**Operation**

IF EFLAGS.VM = 0 (\* Not in Virtual-8086 Mode \*)

THEN IF CPL = 0 OR CR0.PE = 0

THEN

IF OperandSize = 32;

THEN

EFLAGS ← Pop(); (\* 32-bit pop \*)

(\* All non-reserved flags except RF, VIP, VIF, and VM can be modified;  
VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. \*)

ELSE IF (OperandSize = 64)

RFLAGS = Pop(); (\* 64-bit pop \*)

(\* All non-reserved flags except RF, VIP, VIF, and VM can be modified;  
VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. \*)

```

        ELSE (* OperandSize = 16 *)
            EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
            (* All non-reserved flags can be modified. *)
        FI;
    ELSE (* CPL > 0 *)
        IF OperandSize = 32
            THEN
                IF CPL > IOPL
                    THEN
                        EFLAGS ← Pop(); (* 32-bit pop *)
                        (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                        IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                    ELSE
                        EFLAGS ← Pop(); (* 32-bit pop *)
                        (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                        IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                    FI;
                ELSE IF (OperandSize = 64)
                    IF CPL > IOPL
                        THEN
                            RFLAGS ← Pop(); (* 64-bit pop *)
                            (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                            IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                        ELSE
                            RFLAGS ← Pop(); (* 64-bit pop *)
                            (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                            IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                        FI;
                    ELSE (* OperandSize = 16 *)
                        EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
                        (* All non-reserved bits except IOPL can be modified; IOPL and all
                        reserved bits are unaffected. *)
                    FI;
                FI;
            ELSE (* In virtual-8086 mode *)
                IF IOPL = 3
                    THEN
                        IF OperandSize = 32
                            THEN
                                EFLAGS ← Pop();
                                (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                                VIP, VIF, VM, IOPL and all reserved bits are unaffected. RF is cleared. *)
                            ELSE
                                EFLAGS[15:0] ← Pop(); FI;
                                (* All non-reserved bits except IOPL can be modified; IOPL and all reserved bits are unaffected. *)
                            FI;
                        ELSE (* IOPL < 3 *)
                            IF (OperandSize = 32) OR (CR4.VME = 0)
                                THEN #GP(0); (* Trap to virtual-8086 monitor. *)
                            ELSE (* OperandSize = 16 and CR4.VME = 1 *)
                                tempFLAGS ← Pop();
                                IF (EFLAGS.VIP = 1 AND tempFLAGS[9] = 1) OR tempFLAGS[8] = 1
                                    THEN #GP(0);
                                ELSE

```

```
EFLAGS.VIF ← tempEFLAGS[9];
EFLAGS[15:0] ← tempEFLAGS;
(* All non-reserved bits except IOPL and IF can be modified;
IOPL, IF, and all reserved bits are unaffected. *)
```

FI;

Fl;

FI;

Fl;

## Flags Affected

All flags may be affected; see the Operation section for details.

## Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If IOPL < 3 and VME is not enabled. If IOPL < 3 and the 32-bit operand size is used. If IOPL < 3, EFLAGS.VIP = 1, and bit 9 (IF) is set in the FLAGS value on the stack. If IOPL < 3 and bit 8 (TF) is set in the FLAGS value on the stack. If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same as for protected mode exceptions.

## 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF EB /r <sup>1</sup> POR <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Bitwise OR of <i>mm/m64</i> and <i>mm</i> .
66 OF EB /r POR <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Bitwise OR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG EB /r VPOR <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Bitwise OR of <i>xmm2/m128</i> and <i>xmm3</i> .
VEX.NDS.256.66.0F.WIG EB /r VPOR <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Bitwise OR of <i>ymm2/m256</i> and <i>ymm3</i> .
EVEX.NDS.128.66.0F.W0 EB /r VPORD <i>xmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	C	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in <i>xmm2</i> and <i>xmm3/m128/m32bcst</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W0 EB /r VPORD <i>ymm1</i> { <i>k1</i> }[ <i>z</i> ], <i>ymm2</i> , <i>ymm3/m256/m32bcst</i>	C	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in <i>ymm2</i> and <i>ymm3/m256/m32bcst</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W0 EB /r VPORD <i>zmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>zmm2</i> , <i>zmm3/m512/m32bcst</i>	C	V/V	AVX512F	Bitwise OR of packed doubleword integers in <i>zmm2</i> and <i>zmm3/m512/m32bcst</i> using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W1 EB /r VPORQ <i>xmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>xmm2</i> , <i>xmm3/m128/m64bcst</i>	C	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in <i>xmm2</i> and <i>xmm3/m128/m64bcst</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W1 EB /r VPORQ <i>ymm1</i> { <i>k1</i> }[ <i>z</i> ], <i>ymm2</i> , <i>ymm3/m256/m64bcst</i>	C	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in <i>ymm2</i> and <i>ymm3/m256/m64bcst</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W1 EB /r VPORQ <i>zmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>zmm2</i> , <i>zmm3/m512/m64bcst</i>	C	V/V	AVX512F	Bitwise OR of packed quadword integers in <i>zmm2</i> and <i>zmm3/m512/m64bcst</i> using writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
B	NA	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
C	Full	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

### Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

EVEX encoded version: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

## Operation

### POR (64-bit operand)

DEST  $\leftarrow$  DEST OR SRC

### POR (128-bit Legacy SSE version)

DEST  $\leftarrow$  DEST OR SRC

DEST[MAXVL-1:128] (Unmodified)

### VPOR (VEX.128 encoded version)

DEST  $\leftarrow$  SRC1 OR SRC2

DEST[MAXVL-1:128]  $\leftarrow$  0

### VPOR (VEX.256 encoded version)

DEST  $\leftarrow$  SRC1 OR SRC2

DEST[MAXVL-1:256]  $\leftarrow$  0

### VPORD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

    i  $\leftarrow$  j \* 32

    IF k1[j] OR \*no writemask\* THEN

        IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

            THEN DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE OR SRC2[31:0]

            ELSE DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE OR SRC2[i+31:i]

        FI;

    ELSE

        IF \*merging-masking\* ; merging-masking

            \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

            DEST[i+31:i]  $\leftarrow$  0

        FI;

    FI;

ENDFOR;

DEST[MAXVL-1:VL]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPORD \_\_m512i \_mm512\_or\_epi32(\_\_m512i a, \_\_m512i b);  
 VPORD \_\_m512i \_mm512\_mask\_or\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPORD \_\_m512i \_mm512\_maskz\_or\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPORD \_\_m256i \_mm256\_or\_epi32(\_\_m256i a, \_\_m256i b);  
 VPORD \_\_m256i \_mm256\_mask\_or\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORD \_\_m256i \_mm256\_maskz\_or\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORD \_\_m128i \_mm\_or\_epi32(\_\_m128i a, \_\_m128i b);  
 VPORD \_\_m128i \_mm\_mask\_or\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPORD \_\_m128i \_mm\_maskz\_or\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPORQ \_\_m512i \_mm512\_or\_epi64(\_\_m512i a, \_\_m512i b);  
 VPORQ \_\_m512i \_mm512\_mask\_or\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPORQ \_\_m512i \_mm512\_maskz\_or\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPORQ \_\_m256i \_mm256\_or\_epi64(\_\_m256i a, int imm);  
 VPORQ \_\_m256i \_mm256\_mask\_or\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORQ \_\_m256i \_mm256\_maskz\_or\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPORQ \_\_m128i \_mm\_or\_epi64(\_\_m128i a, \_\_m128i b);  
 VPORQ \_\_m128i \_mm\_mask\_or\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPORQ \_\_m128i \_mm\_maskz\_or\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 POR \_\_m64 \_mm\_or\_si64(\_\_m64 m1, \_\_m64 m2)  
 (V)POR: \_\_m128i \_mm\_or\_si128(\_\_m128i m1, \_\_m128i m2)  
 VPOR: \_\_m256i \_mm256\_or\_si256 (\_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.



## PREFETCHh—Prefetch Data Into Caches

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 18 /1	PREFETCHT0 <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using T0 hint.
OF 18 /2	PREFETCHT1 <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using T1 hint.
OF 18 /3	PREFETCHT2 <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using T2 hint.
OF 18 /0	PREFETCHNTA <i>m8</i>	M	Valid	Valid	Move data from <i>m8</i> closer to the processor using NTA hint.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
- T1 (temporal data with respect to first level cache misses)—prefetch data into level 2 cache and higher.
- T2 (temporal data with respect to second level cache misses)—prefetch data into level 3 cache and higher, or an implementation-specific choice.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHh instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHh instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHh instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHh instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHh instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

FETCH (*m8*);

### Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_prefetch(char *p, int i)
```

The argument “\*p” gives the address of the byte (and corresponding cache line) to be prefetched. The value “i” gives a constant (\_MM\_HINT\_T0, \_MM\_HINT\_T1, \_MM\_HINT\_T2, or \_MM\_HINT\_NTA) that specifies the type of prefetch operation to be performed.

### Numeric Exceptions

None.

### Exceptions (All Operating Modes)

#UD                      If the LOCK prefix is used.

## PREFETCHW—Prefetch Data into Caches in Anticipation of a Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 0D /1 PREFETCHW m8	A	V/V	PRFCHW	Move data from m8 closer to the processor in anticipation of a write.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Fetches the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates other cached instances of the line.

The source operand is a byte memory location. If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates other cached copies in anticipation of the line being written to in the future.

The characteristic of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHW instructions, or any other general instruction.

It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

FETCH\_WITH\_EXCLUSIVE\_OWNERSHIP (m8);

### Flags Affected

All flags are affected

### C/C++ Compiler Intrinsic Equivalent

```
void _m_prefetchw( void * );
```

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#UD                      If the LOCK prefix is used.

**Compatibility Mode Exceptions**

#UD                      If the LOCK prefix is used.

**64-Bit Mode Exceptions**

#UD                      If the LOCK prefix is used.

## PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F6 /r <sup>1</sup> PSADBW <i>mm1</i> , <i>mm2</i> /m64	A	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from <i>mm2</i> /m64 and <i>mm1</i> ; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW <i>xmm1</i> , <i>xmm2</i> /m128	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from <i>xmm2</i> /m128 and <i>xmm1</i> ; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.128.66.0F.WIG F6 /r VPSADBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i> /m128	B	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from <i>xmm3</i> /m128 and <i>xmm2</i> ; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.256.66.0F.WIG F6 /r VPSADBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3</i> /m256	B	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from <i>ymm3</i> /m256 and <i>ymm2</i> ; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.128.66.0F.WIG F6 /r VPSADBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i> /m128	C	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from <i>xmm3</i> /m128 and <i>xmm2</i> ; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.256.66.0F.WIG F6 /r VPSADBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3</i> /m256	C	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from <i>ymm3</i> /m256 and <i>ymm2</i> ; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.512.66.0F.WIG F6 /r VPSADBW <i>zmm1</i> , <i>zmm2</i> , <i>zmm3</i> /m512	C	V/V	AVX512BW	Computes the absolute differences of the packed unsigned byte integers from <i>zmm3</i> /m512 and <i>zmm2</i> ; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

## Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-14 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

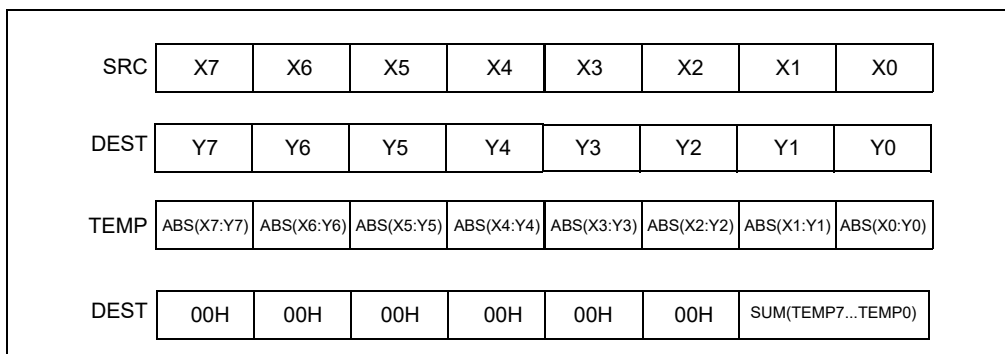
Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.



**Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands**

**Operation****VPSADBW (EVEX encoded versions)**

VL = 128, 256, 512

TEMP0  $\leftarrow$  ABS(SRC1[7:0] - SRC2[7:0])

(\* Repeat operation for bytes 1 through 15 \*)

TEMP15  $\leftarrow$  ABS(SRC1[127:120] - SRC2[127:120])DEST[15:0]  $\leftarrow$  SUM(TEMP0:TEMP7)DEST[63:16]  $\leftarrow$  00000000000000HDEST[79:64]  $\leftarrow$  SUM(TEMP8:TEMP15)DEST[127:80]  $\leftarrow$  00000000000000HIF VL  $\geq$  256

(\* Repeat operation for bytes 16 through 31 \*)

TEMP31  $\leftarrow$  ABS(SRC1[255:248] - SRC2[255:248])DEST[143:128]  $\leftarrow$  SUM(TEMP16:TEMP23)DEST[191:144]  $\leftarrow$  00000000000000HDEST[207:192]  $\leftarrow$  SUM(TEMP24:TEMP31)DEST[223:208]  $\leftarrow$  00000000000000H

FI;

IF VL  $\geq$  512

(\* Repeat operation for bytes 32 through 63 \*)

TEMP63  $\leftarrow$  ABS(SRC1[511:504] - SRC2[511:504])DEST[271:256]  $\leftarrow$  SUM(TEMP0:TEMP7)DEST[319:272]  $\leftarrow$  00000000000000HDEST[335:320]  $\leftarrow$  SUM(TEMP8:TEMP15)DEST[383:336]  $\leftarrow$  00000000000000HDEST[399:384]  $\leftarrow$  SUM(TEMP16:TEMP23)DEST[447:400]  $\leftarrow$  00000000000000HDEST[463:448]  $\leftarrow$  SUM(TEMP24:TEMP31)DEST[511:464]  $\leftarrow$  00000000000000H

FI;

DEST[MAXVL-1:VL]  $\leftarrow$  0**VPSADBW (VEX.256 encoded version)**TEMP0  $\leftarrow$  ABS(SRC1[7:0] - SRC2[7:0])

(\* Repeat operation for bytes 2 through 30 \*)

TEMP31  $\leftarrow$  ABS(SRC1[255:248] - SRC2[255:248])DEST[15:0]  $\leftarrow$  SUM(TEMP0:TEMP7)DEST[63:16]  $\leftarrow$  00000000000000HDEST[79:64]  $\leftarrow$  SUM(TEMP8:TEMP15)DEST[127:80]  $\leftarrow$  00000000000000HDEST[143:128]  $\leftarrow$  SUM(TEMP16:TEMP23)DEST[191:144]  $\leftarrow$  00000000000000HDEST[207:192]  $\leftarrow$  SUM(TEMP24:TEMP31)DEST[223:208]  $\leftarrow$  00000000000000HDEST[MAXVL-1:256]  $\leftarrow$  0

**VPSADBW (VEX.128 encoded version)**

TEMP0  $\leftarrow$  ABS(SRC1[7:0] - SRC2[7:0])

(\* Repeat operation for bytes 2 through 14 \*)

TEMP15  $\leftarrow$  ABS(SRC1[127:120] - SRC2[127:120])

DEST[15:0]  $\leftarrow$  SUM(TEMP0:TEMP7)

DEST[63:16]  $\leftarrow$  0000000000000H

DEST[79:64]  $\leftarrow$  SUM(TEMP8:TEMP15)

DEST[127:80]  $\leftarrow$  0000000000000H

DEST[MAXVL-1:128]  $\leftarrow$  0

**PSADBW (128-bit Legacy SSE version)**

TEMP0  $\leftarrow$  ABS(DEST[7:0] - SRC[7:0])

(\* Repeat operation for bytes 2 through 14 \*)

TEMP15  $\leftarrow$  ABS(DEST[127:120] - SRC[127:120])

DEST[15:0]  $\leftarrow$  SUM(TEMP0:TEMP7)

DEST[63:16]  $\leftarrow$  0000000000000H

DEST[79:64]  $\leftarrow$  SUM(TEMP8:TEMP15)

DEST[127:80]  $\leftarrow$  0000000000000

DEST[MAXVL-1:128] (Unmodified)

**PSADBW (64-bit operand)**

TEMP0  $\leftarrow$  ABS(DEST[7:0] - SRC[7:0])

(\* Repeat operation for bytes 2 through 6 \*)

TEMP7  $\leftarrow$  ABS(DEST[63:56] - SRC[63:56])

DEST[15:0]  $\leftarrow$  SUM(TEMP0:TEMP7)

DEST[63:16]  $\leftarrow$  0000000000000H

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSADBW \_\_m512i \_mm512\_sad\_epu8( \_\_m512i a, \_\_m512i b)

PSADBW: \_\_m64 \_mm\_sad\_pu8( \_\_m64 a, \_\_m64 b)

(V)PSADBW: \_\_m128i \_mm\_sad\_epu8( \_\_m128i a, \_\_m128i b)

VPSADBW: \_\_m256i \_mm256\_sad\_epu8( \_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.



## PSHUFB — Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 38 00 /r <sup>1</sup> PSHUFB <i>mm1</i> , <i>mm2/m64</i>	A	V/V	SSSE3	Shuffle bytes in <i>mm1</i> according to contents of <i>mm2/m64</i> .
66 OF 38 00 /r PSHUFB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSSE3	Shuffle bytes in <i>xmm1</i> according to contents of <i>xmm2/m128</i> .
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Shuffle bytes in <i>xmm2</i> according to contents of <i>xmm3/m128</i> .
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Shuffle bytes in <i>ymm2</i> according to contents of <i>ymm3/m256</i> .
EVEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB <i>xmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Shuffle bytes in <i>xmm2</i> according to contents of <i>xmm3/m128</i> under write mask <i>k1</i> .
EVEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB <i>ymm1</i> { <i>k1</i> }[ <i>z</i> ], <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Shuffle bytes in <i>ymm2</i> according to contents of <i>ymm3/m256</i> under write mask <i>k1</i> .
EVEX.NDS.512.66.0F38.WIG 00 /r VPSHUFB <i>zmm1</i> { <i>k1</i> }[ <i>z</i> ], <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Shuffle bytes in <i>zmm2</i> according to contents of <i>zmm3/m512</i> under write mask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
B	NA	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
C	Full Mem	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

### Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask.

The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four/two in-lane 128-bit shuffles.

## Operation

### PSHUFB (with 64 bit operands)

TEMP  $\leftarrow$  DEST

```
for i = 0 to 7 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0]  $\leftarrow$  0;
    else
        index[2..0]  $\leftarrow$  SRC[(i*8)+2 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0]  $\leftarrow$  TEMP[(index*8+7)..(index*8+0)];
    endif;
}
```

### PSHUFB (with 128 bit operands)

TEMP  $\leftarrow$  DEST

```
for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0]  $\leftarrow$  0;
    else
        index[3..0]  $\leftarrow$  SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0]  $\leftarrow$  TEMP[(index*8+7)..(index*8+0)];
    endif
}
```

### VPSHUFB (VEX.128 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0]  $\leftarrow$  0;
    else
        index[3..0]  $\leftarrow$  SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0]  $\leftarrow$  SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[MAXVL-1:128]  $\leftarrow$  0
```

### VPSHUFB (VEX.256 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7...(i*8)+0]  $\leftarrow$  0;
    else
        index[3..0]  $\leftarrow$  SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0]  $\leftarrow$  SRC1[(index*8+7)..(index*8+0)];
    endif
    if (SRC2[128 + (i * 8)+7] == 1 ) then
        DEST[128 + (i*8)+7...(i*8)+0]  $\leftarrow$  0;
    else
        index[3..0]  $\leftarrow$  SRC2[128 + (i*8)+3 .. (i*8)+0];
        DEST[128 + (i*8)+7...(i*8)+0]  $\leftarrow$  SRC1[128 + (index*8+7)..(index*8+0)];
    endif
}
```

```

endif
}
VPSHUFb (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
jmask ← (KL-1) & ~0xF                                // 0x00, 0x10, 0x30 depending on the VL
FOR j = 0 TO KL-1                                     // dest
    IF kl[ i ] or no_masking
        index ← src.byte[ j ];
        IF index & 0x80
            Dest.byte[ j ] ← 0;
        ELSE
            index ← (index & 0xF) + (j & jmask);        // 16-element in-lane lookup
            Dest.byte[ j ] ← src.byte[ index ];
        ELSE if zeroing
            Dest.byte[ j ] ← 0;
DEST[MAXVL-1:VL] ← 0;

```

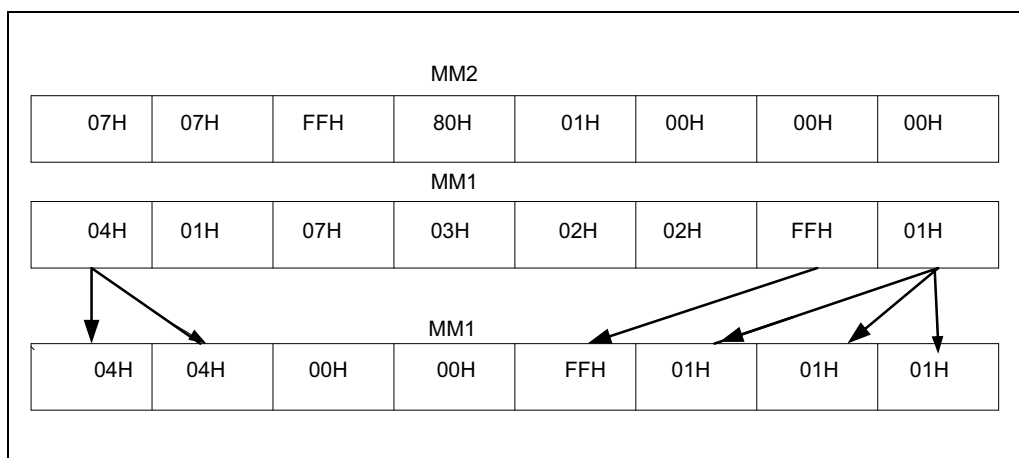


Figure 4-15. PSHUFb with 64-Bit Operands

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPSHUFb __m512i _mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFb __m512i _mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFb __m512i _mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b);
VPSHUFb __m256i _mm256_mask_shuffle_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSHUFb __m256i _mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b);
VPSHUFb __m128i _mm_mask_shuffle_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSHUFb __m128i _mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
PSHUFb: __m64 _mm_shuffle_pi8(__m64 a, __m64 b)
(V)PSHUFb: __m128i _mm_shuffle_epi8(__m128i a, __m128i b)
VPSHUFb: __m256i _mm256_shuffle_epi8(__m256i a, __m256i b)

```

### SIMD Floating-Point Exceptions

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	A	V/V	SSE2	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.66.0F.WIG 70 /r ib VPSHUFD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	A	V/V	AVX	Shuffle the doublewords in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.66.0F.WIG 70 /r ib VPSHUFD <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	A	V/V	AVX2	Shuffle the doublewords in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .
EVEX.128.66.0F.W0 70 /r ib VPSHUFD <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2/m128/m32bcst</i> , <i>imm8</i>	B	V/V	AVX512VL AVX512F	Shuffle the doublewords in <i>xmm2/m128/m32bcst</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.256.66.0F.W0 70 /r ib VPSHUFD <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2/m256/m32bcst</i> , <i>imm8</i>	B	V/V	AVX512VL AVX512F	Shuffle the doublewords in <i>ymm2/m256/m32bcst</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.512.66.0F.W0 70 /r ib VPSHUFD <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2/m512/m32bcst</i> , <i>imm8</i>	B	V/V	AVX512F	Shuffle the doublewords in <i>zmm2/m512/m32bcst</i> based on the encoding in <i>imm8</i> and store the result in <i>zmm1</i> using writemask <i>k1</i> .

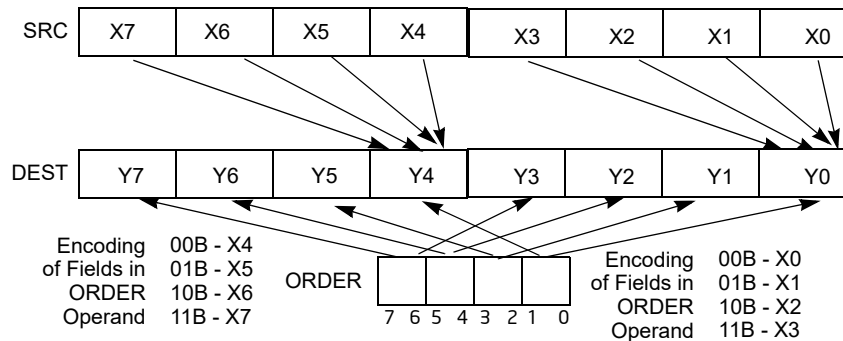
### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-16 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-16) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.



**Figure 4-16. 256-bit VPSHUFD Instruction Operation**

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

In 64-bit mode and not encoded in VEX/EVEX, using REX.R permits this instruction to access XMM8-XMM15.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The source operand can be an YMM register or a 256-bit memory location. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an ZMM/ymm/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/ymm/XMM register updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

Note: EVEX.vvvv and VEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

## Operation

### PSHUFD (128-bit Legacy SSE version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] (Unmodified)
```

### VPSHUFD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] ← 0
```

**VPSHUFD (VEX.256 encoded version)**

```

DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] ← (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] ← (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] ← (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:256] ← 0

```

**VPSHUFD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF (EVEX.b = 1) AND (SRC \*is memory\*)  
         THEN TMP\_SRC[j+31:i] ← SRC[31:0]  
         ELSE TMP\_SRC[j+31:i] ← SRC[i+31:i]

    FI;

ENDFOR;

IF VL >= 128

    TMP\_DEST[31:0] ← (TMP\_SRC[127:0] >> (ORDER[1:0] \* 32))[31:0];  
 TMP\_DEST[63:32] ← (TMP\_SRC[127:0] >> (ORDER[3:2] \* 32))[31:0];  
 TMP\_DEST[95:64] ← (TMP\_SRC[127:0] >> (ORDER[5:4] \* 32))[31:0];  
 TMP\_DEST[127:96] ← (TMP\_SRC[127:0] >> (ORDER[7:6] \* 32))[31:0];

FI;

IF VL >= 256

    TMP\_DEST[159:128] ← (TMP\_SRC[255:128] >> (ORDER[1:0] \* 32))[31:0];  
 TMP\_DEST[191:160] ← (TMP\_SRC[255:128] >> (ORDER[3:2] \* 32))[31:0];  
 TMP\_DEST[223:192] ← (TMP\_SRC[255:128] >> (ORDER[5:4] \* 32))[31:0];  
 TMP\_DEST[255:224] ← (TMP\_SRC[255:128] >> (ORDER[7:6] \* 32))[31:0];

FI;

IF VL >= 512

    TMP\_DEST[287:256] ← (TMP\_SRC[383:256] >> (ORDER[1:0] \* 32))[31:0];  
 TMP\_DEST[319:288] ← (TMP\_SRC[383:256] >> (ORDER[3:2] \* 32))[31:0];  
 TMP\_DEST[351:320] ← (TMP\_SRC[383:256] >> (ORDER[5:4] \* 32))[31:0];  
 TMP\_DEST[383:352] ← (TMP\_SRC[383:256] >> (ORDER[7:6] \* 32))[31:0];  
 TMP\_DEST[415:384] ← (TMP\_SRC[511:384] >> (ORDER[1:0] \* 32))[31:0];  
 TMP\_DEST[447:416] ← (TMP\_SRC[511:384] >> (ORDER[3:2] \* 32))[31:0];  
 TMP\_DEST[479:448] ← (TMP\_SRC[511:384] >> (ORDER[5:4] \* 32))[31:0];  
 TMP\_DEST[511:480] ← (TMP\_SRC[511:384] >> (ORDER[7:6] \* 32))[31:0];

FI;

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\*  
         THEN DEST[j+31:i] ← TMP\_DEST[j+31:i]  
     ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[j+31:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[j+31:i] ← 0

        FI

    FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFD \_\_m512i \_mm512\_shuffle\_epi32(\_\_m512i a, int n);  
 VPSHUFD \_\_m512i \_mm512\_mask\_shuffle\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, int n);  
 VPSHUFD \_\_m512i \_mm512\_maskz\_shuffle\_epi32(\_\_mmask16 k, \_\_m512i a, int n);  
 VPSHUFD \_\_m256i \_mm256\_mask\_shuffle\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, int n);  
 VPSHUFD \_\_m256i \_mm256\_maskz\_shuffle\_epi32(\_\_mmask8 k, \_\_m256i a, int n);  
 VPSHUFD \_\_m128i \_mm\_mask\_shuffle\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, int n);  
 VPSHUFD \_\_m128i \_mm\_maskz\_shuffle\_epi32(\_\_mmask8 k, \_\_m128i a, int n);  
 (V)PSHUFD: \_\_m128i \_mm\_shuffle\_epi32(\_\_m128i a, int n)  
 VPSHUFD: \_\_m256i \_mm256\_shuffle\_epi32(\_\_m256i a, const int n)

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If VEX.vvvv ≠ 1111B or EVEX.vvvv ≠ 1111B.



## PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 70 /r ib PSHUFHW <i>xmm1, xmm2/m128, imm8</i>	A	V/V	SSE2	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW <i>xmm1, xmm2/m128, imm8</i>	A	V/V	AVX	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW <i>ymm1, ymm2/m256, imm8</i>	A	V/V	AVX2	Shuffle the high words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .
EVEX.128.F3.0F.WIG 70 /r ib VPSHUFHW <i>xmm1 {k1}{z}, xmm2/m128, imm8</i>	B	V/V	AVX512VL AVX512BW	Shuffle the high words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> under write mask <i>k1</i> .
EVEX.256.F3.0F.WIG 70 /r ib VPSHUFHW <i>ymm1 {k1}{z}, ymm2/m256, imm8</i>	B	V/V	AVX512VL AVX512BW	Shuffle the high words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> under write mask <i>k1</i> .
EVEX.512.F3.0F.WIG 70 /r ib VPSHUFHW <i>zmm1 {k1}{z}, zmm2/m512, imm8</i>	B	V/V	AVX512BW	Shuffle the high words in <i>zmm2/m512</i> based on the encoding in <i>imm8</i> and store the result in <i>zmm1</i> under write mask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

## Operation

### PSHUFHW (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC[63:0]
DEST[79:64] ← (SRC >> (imm[1:0] * 16))[79:64]
DEST[95:80] ← (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] (Unmodified)
```

### VPSHUFHW (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] * 16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] ← 0
```

### VPSHUFHW (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] * 16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[191:128] ← SRC1[191:128]
DEST[207:192] ← (SRC1 >> (imm[1:0] * 16))[207:192]
DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
DEST[MAXVL-1:256] ← 0
```

### VPSHUFHW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL >= 128

```
TMP_DEST[63:0] ← SRC1[63:0]
TMP_DEST[79:64] ← (SRC1 >> (imm[1:0] * 16))[79:64]
TMP_DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
TMP_DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
TMP_DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
```

FI;

IF VL >= 256

```
TMP_DEST[191:128] ← SRC1[191:128]
TMP_DEST[207:192] ← (SRC1 >> (imm[1:0] * 16))[207:192]
TMP_DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
TMP_DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
TMP_DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
```

FI;

IF VL >= 512

```
TMP_DEST[319:256] ← SRC1[319:256]
TMP_DEST[335:320] ← (SRC1 >> (imm[1:0] * 16))[335:320]
```

```

    TMP_DEST[351:336] ← (SRC1 >> (imm[3:2] * 16))[335:320]
    TMP_DEST[367:352] ← (SRC1 >> (imm[5:4] * 16))[335:320]
    TMP_DEST[383:368] ← (SRC1 >> (imm[7:6] * 16))[335:320]
    TMP_DEST[447:384] ← SRC1[447:384]
    TMP_DEST[463:448] ← (SRC1 >> (imm[1:0] * 16))[463:448]
    TMP_DEST[479:464] ← (SRC1 >> (imm[3:2] * 16))[463:448]
    TMP_DEST[495:480] ← (SRC1 >> (imm[5:4] * 16))[463:448]
    TMP_DEST[511:496] ← (SRC1 >> (imm[7:6] * 16))[463:448]
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i];
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPSHUFWH __m512i __mm512_shufflehi_epi16(__m512i a, int n);
VPSHUFWH __m512i __mm512_mask_shufflehi_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFWH __m512i __mm512_maskz_shufflehi_epi16(__mmask16 k, __m512i a, int n);
VPSHUFWH __m256i __mm256_mask_shufflehi_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFWH __m256i __mm256_maskz_shufflehi_epi16(__mmask8 k, __m256i a, int n);
VPSHUFWH __m128i __mm_mask_shufflehi_epi16(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFWH __m128i __mm_maskz_shufflehi_epi16(__mmask8 k, __m128i a, int n);
(V)PSHUFWH: __m128i __mm_shufflehi_epi16(__m128i a, int n)
VPSHUFWH: __m256i __mm256_shufflehi_epi16(__m256i a, const int n)

```

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

## PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	A	V/V	SSE2	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	A	V/V	AVX	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> .
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	A	V/V	AVX2	Shuffle the low words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> .
EVEX.128.F2.0F.WIG 70 /r ib VPSHUFLW <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2/m128</i> , <i>imm8</i>	B	V/V	AVX512VL AVX512BW	Shuffle the low words in <i>xmm2/m128</i> based on the encoding in <i>imm8</i> and store the result in <i>xmm1</i> under write mask <i>k1</i> .
EVEX.256.F2.0F.WIG 70 /r ib VPSHUFLW <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2/m256</i> , <i>imm8</i>	B	V/V	AVX512VL AVX512BW	Shuffle the low words in <i>ymm2/m256</i> based on the encoding in <i>imm8</i> and store the result in <i>ymm1</i> under write mask <i>k1</i> .
EVEX.512.F2.0F.WIG 70 /r ib VPSHUFLW <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2/m512</i> , <i>imm8</i>	B	V/V	AVX512BW	Shuffle the low words in <i>zmm2/m512</i> based on the encoding in <i>imm8</i> and store the result in <i>zmm1</i> under write mask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

## Operation

### PSHUFLW (128-bit Legacy SSE version)

DEST[15:0]  $\leftarrow$  (SRC  $\gg$  (imm[1:0] \* 16))[15:0]  
 DEST[31:16]  $\leftarrow$  (SRC  $\gg$  (imm[3:2] \* 16))[15:0]  
 DEST[47:32]  $\leftarrow$  (SRC  $\gg$  (imm[5:4] \* 16))[15:0]  
 DEST[63:48]  $\leftarrow$  (SRC  $\gg$  (imm[7:6] \* 16))[15:0]  
 DEST[127:64]  $\leftarrow$  SRC[127:64]  
 DEST[MAXVL-1:128] (Unmodified)

### VPSHUFLW (VEX.128 encoded version)

DEST[15:0]  $\leftarrow$  (SRC1  $\gg$  (imm[1:0] \* 16))[15:0]  
 DEST[31:16]  $\leftarrow$  (SRC1  $\gg$  (imm[3:2] \* 16))[15:0]  
 DEST[47:32]  $\leftarrow$  (SRC1  $\gg$  (imm[5:4] \* 16))[15:0]  
 DEST[63:48]  $\leftarrow$  (SRC1  $\gg$  (imm[7:6] \* 16))[15:0]  
 DEST[127:64]  $\leftarrow$  SRC[127:64]  
 DEST[MAXVL-1:128]  $\leftarrow$  0

### VPSHUFLW (VEX.256 encoded version)

DEST[15:0]  $\leftarrow$  (SRC1  $\gg$  (imm[1:0] \* 16))[15:0]  
 DEST[31:16]  $\leftarrow$  (SRC1  $\gg$  (imm[3:2] \* 16))[15:0]  
 DEST[47:32]  $\leftarrow$  (SRC1  $\gg$  (imm[5:4] \* 16))[15:0]  
 DEST[63:48]  $\leftarrow$  (SRC1  $\gg$  (imm[7:6] \* 16))[15:0]  
 DEST[127:64]  $\leftarrow$  SRC1[127:64]  
 DEST[143:128]  $\leftarrow$  (SRC1  $\gg$  (imm[1:0] \* 16))[143:128]  
 DEST[159:144]  $\leftarrow$  (SRC1  $\gg$  (imm[3:2] \* 16))[143:128]  
 DEST[175:160]  $\leftarrow$  (SRC1  $\gg$  (imm[5:4] \* 16))[143:128]  
 DEST[191:176]  $\leftarrow$  (SRC1  $\gg$  (imm[7:6] \* 16))[143:128]  
 DEST[255:192]  $\leftarrow$  SRC1[255:192]  
 DEST[MAXVL-1:256]  $\leftarrow$  0

### VPSHUFLW (EVEX.U1.512 encoded version)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL  $\geq$  128

TMP\_DEST[15:0]  $\leftarrow$  (SRC1  $\gg$  (imm[1:0] \* 16))[15:0]  
 TMP\_DEST[31:16]  $\leftarrow$  (SRC1  $\gg$  (imm[3:2] \* 16))[15:0]  
 TMP\_DEST[47:32]  $\leftarrow$  (SRC1  $\gg$  (imm[5:4] \* 16))[15:0]  
 TMP\_DEST[63:48]  $\leftarrow$  (SRC1  $\gg$  (imm[7:6] \* 16))[15:0]  
 TMP\_DEST[127:64]  $\leftarrow$  SRC1[127:64]

FI;

IF VL  $\geq$  256

TMP\_DEST[143:128]  $\leftarrow$  (SRC1  $\gg$  (imm[1:0] \* 16))[143:128]  
 TMP\_DEST[159:144]  $\leftarrow$  (SRC1  $\gg$  (imm[3:2] \* 16))[143:128]  
 TMP\_DEST[175:160]  $\leftarrow$  (SRC1  $\gg$  (imm[5:4] \* 16))[143:128]  
 TMP\_DEST[191:176]  $\leftarrow$  (SRC1  $\gg$  (imm[7:6] \* 16))[143:128]  
 TMP\_DEST[255:192]  $\leftarrow$  SRC1[255:192]

FI;

IF VL  $\geq$  512

TMP\_DEST[271:256]  $\leftarrow$  (SRC1  $\gg$  (imm[1:0] \* 16))[271:256]  
 TMP\_DEST[287:272]  $\leftarrow$  (SRC1  $\gg$  (imm[3:2] \* 16))[271:256]  
 TMP\_DEST[303:288]  $\leftarrow$  (SRC1  $\gg$  (imm[5:4] \* 16))[271:256]  
 TMP\_DEST[319:304]  $\leftarrow$  (SRC1  $\gg$  (imm[7:6] \* 16))[271:256]  
 TMP\_DEST[383:320]  $\leftarrow$  SRC1[383:320]

```

    TMP_DEST[399:384] ← (SRC1 >> (imm[1:0] * 16))[399:384]
    TMP_DEST[415:400] ← (SRC1 >> (imm[3:2] * 16))[399:384]
    TMP_DEST[431:416] ← (SRC1 >> (imm[5:4] * 16))[399:384]
    TMP_DEST[447:432] ← (SRC1 >> (imm[7:6] * 16))[399:384]
    TMP_DEST[511:448] ← SRC1[511:448]
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i];
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPSHUFLW __m512i __mm512_shufflelo_epi16(__m512i a, int n);
VPSHUFLW __m512i __mm512_mask_shufflelo_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFLW __m512i __mm512_maskz_shufflelo_epi16(__mmask16 k, __m512i a, int n);
VPSHUFLW __m256i __mm256_mask_shufflelo_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFLW __m256i __mm256_maskz_shufflelo_epi16(__mmask8 k, __m256i a, int n);
VPSHUFLW __m128i __mm_mask_shufflelo_epi16(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFLW __m128i __mm_maskz_shufflelo_epi16(__mmask8 k, __m128i a, int n);
(V)PSHUFLW: __m128i __mm_shufflelo_epi16(__m128i a, int n)
VPSHUFLW: __m256i __mm256_shufflelo_epi16(__m256i a, const int n)

```

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4NF.nb

#UD If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

## PSHUFW—Shuffle Packed Words

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP OF 70 /r ib PSHUFW <i>mm1</i> , <i>mm2/m64</i> , <i>imm8</i>	RMI	Valid	Valid	Shuffle the words in <i>mm2/m64</i> based on the encoding in <i>imm8</i> and store the result in <i>mm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

### Description

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-16. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the source operand to be copied to more than one word location in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

### Operation

```
DEST[15:0] ← (SRC >> (ORDER[1:0] * 16))[15:0];
DEST[31:16] ← (SRC >> (ORDER[3:2] * 16))[15:0];
DEST[47:32] ← (SRC >> (ORDER[5:4] * 16))[15:0];
DEST[63:48] ← (SRC >> (ORDER[7:6] * 16))[15:0];
```

### Intel C/C++ Compiler Intrinsic Equivalent

PSHUFW: `__m64 _mm_shuffle_pi16(__m64 a, int n)`

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

See Table 22-7, “Exception Conditions for SIMD/MMX Instructions with Memory Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## PSIGNB/PSIGNW/PSIGND — Packed SIGN

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 38 08 /r <sup>1</sup> PSIGNB <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in <i>mm1</i> depending on the corresponding sign in <i>mm2/m64</i> .
66 OF 38 08 /r PSIGNB <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in <i>xmm1</i> depending on the corresponding sign in <i>xmm2/m128</i> .
NP OF 38 09 /r <sup>1</sup> PSIGNW <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in <i>mm1</i> depending on the corresponding sign in <i>mm2/m128</i> .
66 OF 38 09 /r PSIGNW <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in <i>xmm1</i> depending on the corresponding sign in <i>xmm2/m128</i> .
NP OF 38 0A /r <sup>1</sup> PSIGND <i>mm1</i> , <i>mm2/m64</i>	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in <i>mm1</i> depending on the corresponding sign in <i>mm2/m128</i> .
66 OF 38 0A /r PSIGND <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in <i>xmm1</i> depending on the corresponding sign in <i>xmm2/m128</i> .
VEX.NDS.128.66.0F38.WIG 08 /r VPSIGNB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Negate/zero/preserve packed byte integers in <i>xmm2</i> depending on the corresponding sign in <i>xmm3/m128</i> .
VEX.NDS.128.66.0F38.WIG 09 /r VPSIGNW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Negate/zero/preserve packed word integers in <i>xmm2</i> depending on the corresponding sign in <i>xmm3/m128</i> .
VEX.NDS.128.66.0F38.WIG 0A /r VPSIGND <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	RVM	V/V	AVX	Negate/zero/preserve packed doubleword integers in <i>xmm2</i> depending on the corresponding sign in <i>xmm3/m128</i> .
VEX.NDS.256.66.0F38.WIG 08 /r VPSIGNB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Negate packed byte integers in <i>ymm2</i> if the corresponding sign in <i>ymm3/m256</i> is less than zero.
VEX.NDS.256.66.0F38.WIG 09 /r VPSIGNW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Negate packed 16-bit integers in <i>ymm2</i> if the corresponding sign in <i>ymm3/m256</i> is less than zero.
VEX.NDS.256.66.0F38.WIG 0A /r VPSIGND <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	RVM	V/V	AVX2	Negate packed doubleword integers in <i>ymm2</i> if the corresponding sign in <i>ymm3/m256</i> is less than zero.
<b>NOTES:</b> 1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A</i> and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the <i>Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A</i> .				

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA



## Description

(V)PSIGNB/(V)PSIGNW/(V)PSIGND negates each data element of the destination operand (the first operand) if the signed integer value of the corresponding data element in the source operand (the second operand) is less than zero. If the signed integer value of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

(V)PSIGNB operates on signed bytes. (V)PSIGNW operates on 16-bit signed words. (V)PSIGND operates on signed 32-bit integers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE instructions: Both operands can be MMX registers. In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

## Operation

### PSIGNB (with 64 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 7th bytes
```

```
IF (SRC[63:56] < 0 )
    DEST[63:56] ← Neg(DEST[63:56])
ELSEIF (SRC[63:56] = 0 )
    DEST[63:56] ← 0
ELSEIF (SRC[63:56] > 0 )
    DEST[63:56] ← DEST[63:56]
```

### PSIGNB (with 128 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 15th bytes
IF (SRC[127:120] < 0 )
    DEST[127:120] ← Neg(DEST[127:120])
ELSEIF (SRC[127:120] = 0 )
    DEST[127:120] ← 0
ELSEIF (SRC[127:120] > 0 )
    DEST[127:120] ← DEST[127:120]
```

**VPSIGNB (VEX.128 encoded version)**

$$\text{DEST}[127:0] \leftarrow \text{BYTE\_SIGN}(\text{SRC1}, \text{SRC2})$$

$$\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$$
**VPSIGNB (VEX.256 encoded version)**

$$\text{DEST}[255:0] \leftarrow \text{BYTE\_SIGN\_256b}(\text{SRC1}, \text{SRC2})$$
**PSIGNW (with 64 bit operands)**

$$\text{IF } (\text{SRC}[15:0] < 0)$$

$$\text{DEST}[15:0] \leftarrow \text{Neg}(\text{DEST}[15:0])$$

$$\text{ELSEIF } (\text{SRC}[15:0] = 0)$$

$$\text{DEST}[15:0] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[15:0] > 0)$$

$$\text{DEST}[15:0] \leftarrow \text{DEST}[15:0]$$

Repeat operation for 2nd through 3rd words

$$\text{IF } (\text{SRC}[63:48] < 0)$$

$$\text{DEST}[63:48] \leftarrow \text{Neg}(\text{DEST}[63:48])$$

$$\text{ELSEIF } (\text{SRC}[63:48] = 0)$$

$$\text{DEST}[63:48] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[63:48] > 0)$$

$$\text{DEST}[63:48] \leftarrow \text{DEST}[63:48]$$
**PSIGNW (with 128 bit operands)**

$$\text{IF } (\text{SRC}[15:0] < 0)$$

$$\text{DEST}[15:0] \leftarrow \text{Neg}(\text{DEST}[15:0])$$

$$\text{ELSEIF } (\text{SRC}[15:0] = 0)$$

$$\text{DEST}[15:0] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[15:0] > 0)$$

$$\text{DEST}[15:0] \leftarrow \text{DEST}[15:0]$$

Repeat operation for 2nd through 7th words

$$\text{IF } (\text{SRC}[127:112] < 0)$$

$$\text{DEST}[127:112] \leftarrow \text{Neg}(\text{DEST}[127:112])$$

$$\text{ELSEIF } (\text{SRC}[127:112] = 0)$$

$$\text{DEST}[127:112] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[127:112] > 0)$$

$$\text{DEST}[127:112] \leftarrow \text{DEST}[127:112]$$
**VPSIGNW (VEX.128 encoded version)**

$$\text{DEST}[127:0] \leftarrow \text{WORD\_SIGN}(\text{SRC1}, \text{SRC2})$$

$$\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$$
**VPSIGNW (VEX.256 encoded version)**

$$\text{DEST}[255:0] \leftarrow \text{WORD\_SIGN}(\text{SRC1}, \text{SRC2})$$
**PSIGND (with 64 bit operands)**

$$\text{IF } (\text{SRC}[31:0] < 0)$$

$$\text{DEST}[31:0] \leftarrow \text{Neg}(\text{DEST}[31:0])$$

$$\text{ELSEIF } (\text{SRC}[31:0] = 0)$$

$$\text{DEST}[31:0] \leftarrow 0$$

$$\text{ELSEIF } (\text{SRC}[31:0] > 0)$$

$$\text{DEST}[31:0] \leftarrow \text{DEST}[31:0]$$

$$\text{IF } (\text{SRC}[63:32] < 0)$$

$$\text{DEST}[63:32] \leftarrow \text{Neg}(\text{DEST}[63:32])$$

$$\text{ELSEIF } (\text{SRC}[63:32] = 0)$$

$$\text{DEST}[63:32] \leftarrow 0$$

```
ELSEIF (SRC[63:32] > 0 )
    DEST[63:32] ← DEST[63:32]
```

**PSIGND (with 128 bit operands)**

```
IF (SRC[31:0] < 0 )
    DEST[31:0] ← Neg(DEST[31:0])
ELSEIF (SRC[31:0] = 0 )
    DEST[31:0] ← 0
ELSEIF (SRC[31:0] > 0 )
    DEST[31:0] ← DEST[31:0]
Repeat operation for 2nd through 3rd double words
IF (SRC[127:96] < 0 )
    DEST[127:96] ← Neg(DEST[127:96])
ELSEIF (SRC[127:96] = 0 )
    DEST[127:96] ← 0
ELSEIF (SRC[127:96] > 0 )
    DEST[127:96] ← DEST[127:96]
```

**VPSIGND (VEX.128 encoded version)**

```
DEST[127:0] ← DWORD_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0
```

**VPSIGND (VEX.256 encoded version)**

```
DEST[255:0] ← DWORD_SIGN(SRC1, SRC2)
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```
PSIGNB:      __m64 _mm_sign_pi8 (__m64 a, __m64 b)
(V)PSIGNB:   __m128i _mm_sign_epi8 (__m128i a, __m128i b)
VPSIGNB:     __m256i _mm256_sign_epi8 (__m256i a, __m256i b)
PSIGNW:      __m64 _mm_sign_pi16 (__m64 a, __m64 b)
(V)PSIGNW:   __m128i _mm_sign_epi16 (__m128i a, __m128i b)
VPSIGNW:     __m256i _mm256_sign_epi16 (__m256i a, __m256i b)
PSIGND:      __m64 _mm_sign_pi32 (__m64 a, __m64 b)
(V)PSIGND:   __m128i _mm_sign_epi32 (__m128i a, __m128i b)
VPSIGND:     __m256i _mm256_sign_epi32 (__m256i a, __m256i b)
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.L = 1.

## PSLLDQ—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ <i>xmm1</i> , <i>imm8</i>	A	V/V	SSE2	Shift <i>xmm1</i> left by <i>imm8</i> bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	B	V/V	AVX	Shift <i>xmm2</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>xmm1</i> .
VEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	B	V/V	AVX2	Shift <i>ymm2</i> left by <i>imm8</i> bytes while shifting in 0s and store result in <i>ymm1</i> .
EVEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ <i>xmm1</i> , <i>xmm2</i> /m128, <i>imm8</i>	C	V/V	AVX512VL AVX512BW	Shift <i>xmm2</i> /m128 left by <i>imm8</i> bytes while shifting in 0s and store result in <i>xmm1</i> .
EVEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ <i>ymm1</i> , <i>ymm2</i> /m256, <i>imm8</i>	C	V/V	AVX512VL AVX512BW	Shift <i>ymm2</i> /m256 left by <i>imm8</i> bytes while shifting in 0s and store result in <i>ymm1</i> .
EVEX.NDD.512.66.0F.WIG 73 /7 ib VPSLLDQ <i>zmm1</i> , <i>zmm2</i> /m512, <i>imm8</i>	C	V/V	AVX512BW	Shift <i>zmm2</i> /m512 left by <i>imm8</i> bytes while shifting in 0s and store result in <i>zmm1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r, w)	imm8	NA	NA
B	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA

### Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

### Operation

**VPSLLDQ (EVEX.U1.512 encoded version)**

TEMP ← COUNT

IF (TEMP > 15) THEN TEMP ← 16; FI

DEST[127:0] ← SRC[127:0] << (TEMP \* 8)

DEST[255:128] ← SRC[255:128] << (TEMP \* 8)

DEST[383:256] ← SRC[383:256] << (TEMP \* 8)

DEST[511:384] ← SRC[511:384] << (TEMP \* 8)

DEST[MAXVL-1:512] ← 0

**VPSLLDQ (VEX.256 and EVEX.256 encoded version)**TEMP  $\leftarrow$  COUNTIF (TEMP > 15) THEN TEMP  $\leftarrow$  16; FIDEST[127:0]  $\leftarrow$  SRC[127:0]  $\ll$  (TEMP \* 8)DEST[255:128]  $\leftarrow$  SRC[255:128]  $\ll$  (TEMP \* 8)DEST[MAXVL-1:256]  $\leftarrow$  0**VPSLLDQ (VEX.128 and EVEX.128 encoded version)**TEMP  $\leftarrow$  COUNTIF (TEMP > 15) THEN TEMP  $\leftarrow$  16; FIDEST  $\leftarrow$  SRC  $\ll$  (TEMP \* 8)DEST[MAXVL-1:128]  $\leftarrow$  0**PSLLDQ(128-bit Legacy SSE version)**TEMP  $\leftarrow$  COUNTIF (TEMP > 15) THEN TEMP  $\leftarrow$  16; FIDEST  $\leftarrow$  DEST  $\ll$  (TEMP \* 8)

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

(V)PSLLDQ: \_\_m128i \_mm\_slli\_si128 ( \_\_m128i a, int imm)

VPSLLDQ: \_\_m256i \_mm256\_slli\_si256 ( \_\_m256i a, const int imm)

VPSLLDQ \_\_m512i \_mm512\_bslli\_epi128 ( \_\_m512i a, const int imm)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF F1 /r <sup>1</sup> PSLLW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Shift words in <i>mm</i> left <i>mm/m64</i> while shifting in 0s.
66 OF F1 /r PSLLW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Shift words in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
NP OF 71 /6 ib PSLLW <i>mm1</i> , <i>imm8</i>	B	V/V	MMX	Shift words in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 OF 71 /6 ib PSLLW <i>xmm1</i> , <i>imm8</i>	B	V/V	SSE2	Shift words in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
NP OF F2 /r <sup>1</sup> PSLLD <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Shift doublewords in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 OF F2 /r PSLLD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Shift doublewords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
NP OF 72 /6 ib <sup>1</sup> PSLLD <i>mm</i> , <i>imm8</i>	B	V/V	MMX	Shift doublewords in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 OF 72 /6 ib PSLLD <i>xmm1</i> , <i>imm8</i>	B	V/V	SSE2	Shift doublewords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
NP OF F3 /r <sup>1</sup> PSLLQ <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Shift quadword in <i>mm</i> left by <i>mm/m64</i> while shifting in 0s.
66 OF F3 /r PSLLQ <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Shift quadwords in <i>xmm1</i> left by <i>xmm2/m128</i> while shifting in 0s.
NP OF 73 /6 ib <sup>1</sup> PSLLQ <i>mm</i> , <i>imm8</i>	B	V/V	MMX	Shift quadword in <i>mm</i> left by <i>imm8</i> while shifting in 0s.
66 OF 73 /6 ib PSLLQ <i>xmm1</i> , <i>imm8</i>	B	V/V	SSE2	Shift quadwords in <i>xmm1</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.OF.WIG F1 /r VPSLLW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX	Shift words in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.OF.WIG 71 /6 ib VPSLLW <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	D	V/V	AVX	Shift words in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.OF.WIG F2 /r VPSLLD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX	Shift doublewords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.OF.WIG 72 /6 ib VPSLLD <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	D	V/V	AVX	Shift doublewords in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.OF.WIG F3 /r VPSLLQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX	Shift quadwords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.OF.WIG 73 /6 ib VPSLLQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	D	V/V	AVX	Shift quadwords in <i>xmm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.OF.WIG F1 /r VPSLLW <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	C	V/V	AVX2	Shift words in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.OF.WIG 71 /6 ib VPSLLW <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	D	V/V	AVX2	Shift words in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.

VEX.NDS.256.66.0F.WIG F2 /r VPSLLD <i>ymm1, ymm2, xmm3/m128</i>	C	V/V	AVX2	Shift doublewords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD <i>ymm1, ymm2, imm8</i>	D	V/V	AVX2	Shift doublewords in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ <i>ymm1, ymm2, xmm3/m128</i>	C	V/V	AVX2	Shift quadwords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ <i>ymm1, ymm2, imm8</i>	D	V/V	AVX2	Shift quadwords in <i>ymm2</i> left by <i>imm8</i> while shifting in 0s.
EVEX.NDS.128.66.0F.WIG F1 /r VPSLLW <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	G	V/V	AVX512VL AVX512BW	Shift words in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG F1 /r VPSLLW <i>ymm1 {k1}{z}, ymm2, xmm3/m128</i>	G	V/V	AVX512VL AVX512BW	Shift words in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG F1 /r VPSLLW <i>zmm1 {k1}{z}, zmm2, xmm3/m128</i>	G	V/V	AVX512BW	Shift words in <i>zmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW <i>xmm1 {k1}{z}, xmm2/m128, imm8</i>	E	V/V	AVX512VL AVX512BW	Shift words in <i>xmm2/m128</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW <i>ymm1 {k1}{z}, ymm2/m256, imm8</i>	E	V/V	AVX512VL AVX512BW	Shift words in <i>ymm2/m256</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.512.66.0F.WIG 71 /6 ib VPSLLW <i>zmm1 {k1}{z}, zmm2/m512, imm8</i>	E	V/V	AVX512BW	Shift words in <i>zmm2/m512</i> left by <i>imm8</i> while shifting in 0 using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W0 F2 /r VPSLLD <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	G	V/V	AVX512VL AVX512F	Shift doublewords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s under writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W0 F2 /r VPSLLD <i>ymm1 {k1}{z}, ymm2, xmm3/m128</i>	G	V/V	AVX512VL AVX512F	Shift doublewords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s under writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W0 F2 /r VPSLLD <i>zmm1 {k1}{z}, zmm2, xmm3/m128</i>	G	V/V	AVX512F	Shift doublewords in <i>zmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s under writemask <i>k1</i> .
EVEX.NDD.128.66.0F.W0 72 /6 ib VPSLLD <i>xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</i>	F	V/V	AVX512VL AVX512F	Shift doublewords in <i>xmm2/m128/m32bcst</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.256.66.0F.W0 72 /6 ib VPSLLD <i>ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</i>	F	V/V	AVX512VL AVX512F	Shift doublewords in <i>ymm2/m256/m32bcst</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.512.66.0F.W0 72 /6 ib VPSLLD <i>zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8</i>	F	V/V	AVX512F	Shift doublewords in <i>zmm2/m512/m32bcst</i> left by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W1 F3 /r VPSLLQ <i>xmm1 {k1}{z}, xmm2, xmm3/m128</i>	G	V/V	AVX512VL AVX512F	Shift quadwords in <i>xmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W1 F3 /r VPSLLQ <i>ymm1 {k1}{z}, ymm2, xmm3/m128</i>	G	V/V	AVX512VL AVX512F	Shift quadwords in <i>ymm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W1 F3 /r VPSLLQ <i>zmm1 {k1}{z}, zmm2, xmm3/m128</i>	G	V/V	AVX512F	Shift quadwords in <i>zmm2</i> left by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .

EVEX.NDD.128.66.0F.W1 73 /6 ib VPSLLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /6 ib VPSLLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

**NOTES:**

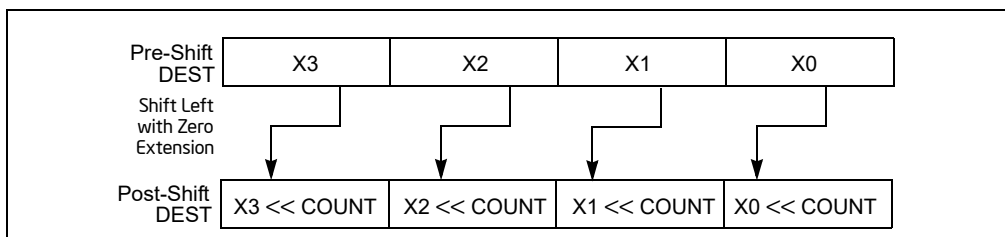
1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand.



**Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand**

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.



128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

## Operation

### PSLLW (with 64-bit operand)

```
IF (COUNT > 15)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
  FI;
```

### PSLLD (with 64-bit operand)

```
IF (COUNT > 31)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
  FI;
```

### PSLLQ (with 64-bit operand)

```
IF (COUNT > 63)
  THEN
    DEST[64:0] ← 0000000000000000H;
  ELSE
    DEST ← ZeroExtend(DEST << COUNT);
  FI;
```

LOGICAL\_LEFT\_SHIFT\_WORDS(SRC, COUNT\_SRC)

COUNT ← COUNT\_SRC[63:0];

IF (COUNT > 15)

THEN

```

    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(SRC[127:112] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ← ZeroExtend(SRC[127:96] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
    DEST[255:128] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)

```

```

    DEST[255:240] ← ZeroExtend(SRC[255:240] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
    DEST[255:128] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] ← ZeroExtend(SRC[255:224] << COUNT);
FI;

```

```

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
    DEST[255:128] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] << COUNT);
    DEST[191:128] ← ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] ← ZeroExtend(SRC[255:192] << COUNT);
FI;

```

#### **VPSLLW (EVEX versions, xmm/m128)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

```

```

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] = 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPSLLW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_128b(SRC1[127:0], imm8)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

TMP\_DEST[511:256] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1[511:256], imm8)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSLLW (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0;

**VPSLLW (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_WORD\_256b(SRC1, imm8)

DEST[MAXVL-1:256] ← 0;

**VPSLLW (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(SRC1, SRC2)

DEST[MAXVL-1:128] ← 0

**VPSLLW (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(SRC1, imm8)

DEST[MAXVL-1:128] ← 0

**PSLLW (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, SRC)

DEST[MAXVL-1:128] (Unmodified)

**PSLLW (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_WORDS(DEST, imm8)

DEST[MAXVL-1:128] (Unmodified)

**VPSLLD (EVEX versions, imm8)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+31:i] ← LOGICAL\_LEFT\_SHIFT\_DWORDS1(SRC1[31:0], imm8)

ELSE DEST[i+31:i] ← LOGICAL\_LEFT\_SHIFT\_DWORDS1(SRC1[i+31:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSLLD (EVEX versions, xmm/m128)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSLLD (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0;

**VPSLLD (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS\_256b(SRC1, imm8)

DEST[MAXVL-1:256] ← 0;

**VPSLLD (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(SRC1, SRC2)

DEST[MAXVL-1:128] ← 0

**VPSLLD (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(SRC1, imm8)

DEST[MAXVL-1:128] ← 0

**PSLLD (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(DEST, SRC)

DEST[MAXVL-1:128] (Unmodified)

**PSLLD (xmm, imm8)**

DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_DWORDS(DEST, imm8)

DEST[MAXVL-1:128] (Unmodified)

**VPSLLQ (EVEX versions, imm8)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+63:i] ← LOGICAL\_LEFT\_SHIFT\_QWORDS1(SRC1[63:0], imm8)

ELSE DEST[i+63:i] ← LOGICAL\_LEFT\_SHIFT\_QWORDS1(SRC1[i+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

**VPSLLQ (EVEX versions, xmm/m128)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_LEFT\_SHIFT\_QWORDS\_256b(SRC1[511:256], SRC2)

FI;

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 encoding**

```

DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0;

```

**VPSLLQ (ymm, imm8) - VEX.256 encoding**

```

DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0;

```

**VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 encoding**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

```

**VPSLLQ (xmm, imm8) - VEX.128 encoding**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0

```

**PSLLQ (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

```

**PSLLQ (xmm, imm8)**

```

DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPSLLD __m512i __mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i __mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i __mm512_maskz_slli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i __mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i __mm256_maskz_slli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i __mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i __mm_maskz_slli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i __mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i __mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i __mm512_maskz_sll_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i __mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i __mm256_maskz_sll_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i __mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i __mm_maskz_sll_epi32(__mmask8 k, __m128i a, __m128i cnt);

```

VPSLLQ \_\_m512i \_mm512\_mask\_slli\_epi64(\_\_m512i a, unsigned int imm);  
 VPSLLQ \_\_m512i \_mm512\_mask\_slli\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, unsigned int imm);  
 VPSLLQ \_\_m512i \_mm512\_maskz\_slli\_epi64(\_\_mmask8 k, \_\_m512i a, unsigned int imm);  
 VPSLLQ \_\_m256i \_mm256\_mask\_slli\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSLLQ \_\_m256i \_mm256\_maskz\_slli\_epi64(\_\_mmask8 k, \_\_m256i a, unsigned int imm);  
 VPSLLQ \_\_m128i \_mm\_mask\_slli\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSLLQ \_\_m128i \_mm\_maskz\_slli\_epi64(\_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSLLQ \_\_m512i \_mm512\_mask\_sll\_epi64(\_\_m512i a, \_\_m128i cnt);  
 VPSLLQ \_\_m512i \_mm512\_mask\_sll\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m128i cnt);  
 VPSLLQ \_\_m512i \_mm512\_maskz\_sll\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m128i cnt);  
 VPSLLQ \_\_m256i \_mm256\_mask\_sll\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSLLQ \_\_m256i \_mm256\_maskz\_sll\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSLLQ \_\_m128i \_mm\_mask\_sll\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLQ \_\_m128i \_mm\_maskz\_sll\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLW \_\_m512i \_mm512\_slli\_epi16(\_\_m512i a, unsigned int imm);  
 VPSLLW \_\_m512i \_mm512\_mask\_slli\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSLLW \_\_m512i \_mm512\_maskz\_slli\_epi16(\_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSLLW \_\_m256i \_mm256\_mask\_slli\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSLLW \_\_m256i \_mm256\_maskz\_slli\_epi16(\_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSLLW \_\_m128i \_mm\_mask\_slli\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSLLW \_\_m128i \_mm\_maskz\_slli\_epi16(\_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSLLW \_\_m512i \_mm512\_sll\_epi16(\_\_m512i a, \_\_m128i cnt);  
 VPSLLW \_\_m512i \_mm512\_mask\_sll\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m128i cnt);  
 VPSLLW \_\_m512i \_mm512\_maskz\_sll\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m128i cnt);  
 VPSLLW \_\_m256i \_mm256\_mask\_sll\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m128i cnt);  
 VPSLLW \_\_m256i \_mm256\_maskz\_sll\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m128i cnt);  
 VPSLLW \_\_m128i \_mm\_mask\_sll\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLW \_\_m128i \_mm\_maskz\_sll\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 PSLLW: \_\_m64 \_mm\_slli\_pi16(\_\_m64 m, int count)  
 PSLLW: \_\_m64 \_mm\_sll\_pi16(\_\_m64 m, \_\_m64 count)  
 (V)PSLLW: \_\_m128i \_mm\_slli\_epi16(\_\_m64 m, int count)  
 (V)PSLLW: \_\_m128i \_mm\_sll\_epi16(\_\_m128i m, \_\_m128i count)  
 VPSLLW: \_\_m256i \_mm256\_slli\_epi16(\_\_m256i m, int count)  
 VPSLLW: \_\_m256i \_mm256\_sll\_epi16(\_\_m256i m, \_\_m128i count)  
 PSLLD: \_\_m64 \_mm\_slli\_pi32(\_\_m64 m, int count)  
 PSLLD: \_\_m64 \_mm\_sll\_pi32(\_\_m64 m, \_\_m64 count)  
 (V)PSLLD: \_\_m128i \_mm\_slli\_epi32(\_\_m128i m, int count)  
 (V)PSLLD: \_\_m128i \_mm\_sll\_epi32(\_\_m128i m, \_\_m128i count)  
 VPSLLD: \_\_m256i \_mm256\_slli\_epi32(\_\_m256i m, int count)  
 VPSLLD: \_\_m256i \_mm256\_sll\_epi32(\_\_m256i m, \_\_m128i count)  
 PSLLQ: \_\_m64 \_mm\_slli\_si64(\_\_m64 m, int count)  
 PSLLQ: \_\_m64 \_mm\_sll\_si64(\_\_m64 m, \_\_m64 count)  
 (V)PSLLQ: \_\_m128i \_mm\_slli\_epi64(\_\_m128i m, int count)  
 (V)PSLLQ: \_\_m128i \_mm\_sll\_epi64(\_\_m128i m, \_\_m128i count)  
 VPSLLQ: \_\_m256i \_mm256\_slli\_epi64(\_\_m256i m, int count)  
 VPSLLQ: \_\_m256i \_mm256\_sll\_epi64(\_\_m256i m, \_\_m128i count)

### Flags Affected

None.

### Numeric Exceptions

None.



## Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.

Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSLLW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSLLD/Q:

Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.

Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.

## PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF E1 /r <sup>1</sup> PSRAW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Shift words in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 OF E1 /r PSRAW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
NP OF 71 /4 ib <sup>1</sup> PSRAW <i>mm</i> , <i>imm8</i>	B	V/V	MMX	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 OF 71 /4 ib PSRAW <i>xmm1</i> , <i>imm8</i>	B	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.
NP OF E2 /r <sup>1</sup> PSRAD <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>mm/m64</i> while shifting in sign bits.
66 OF E2 /r PSRAD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Shift doubleword in <i>xmm1</i> right by <i>xmm2/m128</i> while shifting in sign bits.
NP OF 72 /4 ib <sup>1</sup> PSRAD <i>mm</i> , <i>imm8</i>	B	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in sign bits.
66 OF 72 /4 ib PSRAD <i>xmm1</i> , <i>imm8</i>	B	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.128.66.OF.WIG E1 /r VPSRAW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.128.66.OF.WIG 71 /4 ib VPSRAW <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	D	V/V	AVX	Shift words in <i>xmm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.128.66.OF.WIG E2 /r VPSRAD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.128.66.OF.WIG 72 /4 ib VPSRAD <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	D	V/V	AVX	Shift doublewords in <i>xmm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.256.66.OF.WIG E1 /r VPSRAW <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	C	V/V	AVX2	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.256.66.OF.WIG 71 /4 ib VPSRAW <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	D	V/V	AVX2	Shift words in <i>ymm2</i> right by <i>imm8</i> while shifting in sign bits.
VEX.NDS.256.66.OF.WIG E2 /r VPSRAD <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	C	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits.
VEX.NDD.256.66.OF.WIG 72 /4 ib VPSRAD <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	D	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by <i>imm8</i> while shifting in sign bits.
EVEX.NDS.128.66.OF.WIG E1 /r VPSRAW <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512VL AVX512BW	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits using writemask <i>k1</i> .
EVEX.NDS.256.66.OF.WIG E1 /r VPSRAW <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512VL AVX512BW	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits using writemask <i>k1</i> .
EVEX.NDS.512.66.OF.WIG E1 /r VPSRAW <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512BW	Shift words in <i>zmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in sign bits using writemask <i>k1</i> .

EVEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /4 ib VPSRAW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW	Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F.W0 E2 /r VPSRAD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.W0 E2 /r VPSRAD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.W0 E2 /r VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /4 ib VPSRAD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /4 ib VPSRAD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /4 ib VPSRAD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F.W1 E2 /r VPSRAQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.W1 E2 /r VPSRAQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.W1 E2 /r VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.128.66.0F.W1 72 /4 ib VPSRAQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.W1 72 /4 ib VPSRAQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.W1 72 /4 ib VPSRAQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.

## NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-18 gives an example of shifting words in a 64-bit operand.)

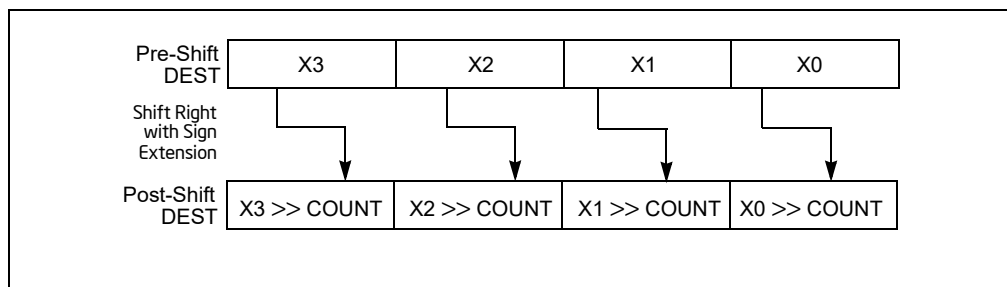


Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

## Operation

### PSRAW (with 64-bit operand)

```
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
DEST[63:48] ← SignExtend(DEST[63:48] >> COUNT);
```

### PSRAD (with 64-bit operand)

```
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
DEST[63:32] ← SignExtend(DEST[63:32] >> COUNT);
```

ARITHMETIC\_RIGHT\_SHIFT\_DWORDS1(SRC, COUNT\_SRC)

```
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN
        DEST[31:0] ← SignBit
    ELSE
        DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
FI;
```

ARITHMETIC\_RIGHT\_SHIFT\_QWORDS1(SRC, COUNT\_SRC)

```
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN
        DEST[63:0] ← SignBit
    ELSE
        DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
FI;
```

ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC, COUNT\_SRC)

```
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] ← SignExtend(SRC[255:240] >> COUNT);
```

```

ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] ← SignExtend(SRC[255:224] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL)          ; VL: 128b, 256b or 512b
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN    COUNT ← 64;
FI;
DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] ← SignExtend(SRC[VL-1:VL-64] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(SRC[127:112] >> COUNT);

```

```

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
(* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] ← SignExtend(SRC[127:96] >> COUNT);

```

**VPSRAW (EVEX versions, xmm/m128)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSRAW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], imm8)

FI;

IF VL = 256

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

FI;

IF VL = 512

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

TMP\_DEST[511:256] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], imm8)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSRAW (ymm, ymm, xmm/m128) - VEX**

DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0

**VPSRAW (ymm, imm8) - VEX**

DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, imm8)

DEST[MAXVL-1:256] ← 0

**VPSRAW (xmm, xmm, xmm/m128) - VEX**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(SRC1, SRC2)

DEST[MAXVL-1:128] ← 0

**VPSRAW (xmm, imm8) - VEX**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(SRC1, imm8)

DEST[MAXVL-1:128] ← 0

**PSRAW (xmm, xmm, xmm/m128)**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(DEST, SRC)

DEST[MAXVL-1:128] (Unmodified)

**PSRAW (xmm, imm8)**

DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_WORDS(DEST, imm8)

DEST[MAXVL-1:128] (Unmodified)

**VPSRAD (EVEX versions, imm8)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+31:i] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS1(SRC1[31:0], imm8)

ELSE DEST[i+31:i] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS1(SRC1[i+31:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSRAD (EVEX versions, xmm/m128)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

TMP\_DEST[127:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← ARITHMETIC\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[511:256], SRC2)



```

FI;

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPSRAD (ymm, ymm, xmm/m128) - VEX**

```

DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

```

**VPSRAD (ymm, imm8) - VEX**

```

DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0

```

**VPSRAD (xmm, xmm, xmm/m128) - VEX**

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

```

**VPSRAD (xmm, imm8) - VEX**

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0

```

**PSRAD (xmm, xmm, xmm/m128)**

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

```

**PSRAD (xmm, imm8)**

```

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)

```

**VPSRAQ (EVEX versions, imm8)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
    ELSE DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
  FI;
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
  ELSE *zeroing-masking* ; zeroing-masking
    DEST[i+63:i] ← 0
  FI;
ENDFOR

```

```

        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

### VPSRAQ (EVEX versions, xmm/m128)

(KL, VL) = (2, 128), (4, 256), (8, 512)

TMP\_DEST[VL-1:0] ← ARITHMETIC\_RIGHT\_SHIFT\_QWORDS(SRC1[VL-1:0], SRC2, VL)

```

FOR j ← 0 TO 7
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPSRAD __m512i _mm512_srai_epi32(__m512i a, unsigned int imm);
VPSRAD __m512i _mm512_mask_srai_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m512i _mm512_maskz_srai_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m256i _mm256_mask_srai_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m256i _mm256_maskz_srai_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m128i _mm_mask_srai_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m128i _mm_maskz_srai_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m512i _mm512_sra_epi32(__m512i a, __m128i cnt);
VPSRAD __m512i _mm512_mask_sra_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m512i _mm512_maskz_sra_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m256i _mm256_mask_sra_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m256i _mm256_maskz_sra_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m128i _mm_mask_sra_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAD __m128i _mm_maskz_sra_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m512i _mm512_srai_epi64(__m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_mask_srai_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_maskz_srai_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRAQ __m256i _mm256_mask_srai_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m256i _mm256_maskz_srai_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m128i _mm_mask_srai_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m128i _mm_maskz_srai_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m512i _mm512_sra_epi64(__m512i a, __m128i cnt);
VPSRAQ __m512i _mm512_mask_sra_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRAQ __m512i _mm512_maskz_sra_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSRAQ __m256i _mm256_mask_sra_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m256i _mm256_maskz_sra_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m128i _mm_mask_sra_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m128i _mm_maskz_sra_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m512i _mm512_srai_epi16(__m512i a, unsigned int imm);
VPSRAW __m512i _mm512_mask_srai_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);

```

VPSRAW \_\_m512i \_\_mm512\_maskz\_srai\_epi16( \_\_mmask32 k, \_\_m512i a, unsigned int imm);  
 VPSRAW \_\_m256i \_\_mm256\_mask\_srai\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSRAW \_\_m256i \_\_mm256\_maskz\_srai\_epi16( \_\_mmask16 k, \_\_m256i a, unsigned int imm);  
 VPSRAW \_\_m128i \_\_mm\_mask\_srai\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRAW \_\_m128i \_\_mm\_maskz\_srai\_epi16( \_\_mmask8 k, \_\_m128i a, unsigned int imm);  
 VPSRAW \_\_m512i \_\_mm512\_sra\_epi16(\_\_m512i a, \_\_m128i cnt);  
 VPSRAW \_\_m512i \_\_mm512\_mask\_sra\_epi16(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRAW \_\_m512i \_\_mm512\_maskz\_sra\_epi16( \_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);  
 VPSRAW \_\_m256i \_\_mm256\_mask\_sra\_epi16(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRAW \_\_m256i \_\_mm256\_maskz\_sra\_epi16( \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);  
 VPSRAW \_\_m128i \_\_mm\_mask\_sra\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRAW \_\_m128i \_\_mm\_maskz\_sra\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 PSRAW: \_\_m64 \_\_mm\_srai\_pi16 ( \_\_m64 m, int count)  
 PSRAW: \_\_m64 \_\_mm\_sra\_pi16 ( \_\_m64 m, \_\_m64 count)  
 (V)PSRAW: \_\_m128i \_\_mm\_srai\_epi16(\_\_m128i m, int count)  
 (V)PSRAW: \_\_m128i \_\_mm\_sra\_epi16(\_\_m128i m, \_\_m128i count)  
 VPSRAW: \_\_m256i \_\_mm256\_srai\_epi16 ( \_\_m256i m, int count)  
 VPSRAW: \_\_m256i \_\_mm256\_sra\_epi16 ( \_\_m256i m, \_\_m128i count)  
 PSRAD: \_\_m64 \_\_mm\_srai\_pi32 ( \_\_m64 m, int count)  
 PSRAD: \_\_m64 \_\_mm\_sra\_pi32 ( \_\_m64 m, \_\_m64 count)  
 (V)PSRAD: \_\_m128i \_\_mm\_srai\_epi32 ( \_\_m128i m, int count)  
 (V)PSRAD: \_\_m128i \_\_mm\_sra\_epi32 ( \_\_m128i m, \_\_m128i count)  
 VPSRAD: \_\_m256i \_\_mm256\_srai\_epi32 ( \_\_m256i m, int count)  
 VPSRAD: \_\_m256i \_\_mm256\_sra\_epi32 ( \_\_m256i m, \_\_m128i count)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.

Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSRAW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSRAD/Q:

Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.

Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.

## PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ <i>xmm1</i> , <i>imm8</i>	A	V/V	SSE2	Shift <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	B	V/V	AVX	Shift <i>xmm2</i> right by <i>imm8</i> bytes while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	B	V/V	AVX2	Shift <i>ymm1</i> right by <i>imm8</i> bytes while shifting in 0s.
EVEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	C	V/V	AVX512VL AVX512BW	Shift <i>xmm2/m128</i> right by <i>imm8</i> bytes while shifting in 0s and store result in <i>xmm1</i> .
EVEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	C	V/V	AVX512VL AVX512BW	Shift <i>ymm2/m256</i> right by <i>imm8</i> bytes while shifting in 0s and store result in <i>ymm1</i> .
EVEX.NDD.512.66.0F.WIG 73 /3 ib VPSRLDQ <i>zmm1</i> , <i>zmm2/m512</i> , <i>imm8</i>	C	V/V	AVX512BW	Shift <i>zmm2/m512</i> right by <i>imm8</i> bytes while shifting in 0s and store result in <i>zmm1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r, w)	imm8	NA	NA
B	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
C	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA

### Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Note: VEX.vvvv/EVEX.vvvv encodes the destination register.

**Operation****VPSRLDQ (EVEX.512 encoded version)**

$TEMP \leftarrow COUNT$   
 IF  $(TEMP > 15)$  THEN  $TEMP \leftarrow 16$ ; FI  
 $DEST[127:0] \leftarrow SRC[127:0] \gg (TEMP * 8)$   
 $DEST[255:128] \leftarrow SRC[255:128] \gg (TEMP * 8)$   
 $DEST[383:256] \leftarrow SRC[383:256] \gg (TEMP * 8)$   
 $DEST[511:384] \leftarrow SRC[511:384] \gg (TEMP * 8)$   
 $DEST[MAXVL-1:512] \leftarrow 0$ ;

**VPSRLDQ (VEX.256 and EVEX.256 encoded version)**

$TEMP \leftarrow COUNT$   
 IF  $(TEMP > 15)$  THEN  $TEMP \leftarrow 16$ ; FI  
 $DEST[127:0] \leftarrow SRC[127:0] \gg (TEMP * 8)$   
 $DEST[255:128] \leftarrow SRC[255:128] \gg (TEMP * 8)$   
 $DEST[MAXVL-1:256] \leftarrow 0$ ;

**VPSRLDQ (VEX.128 and EVEX.128 encoded version)**

$TEMP \leftarrow COUNT$   
 IF  $(TEMP > 15)$  THEN  $TEMP \leftarrow 16$ ; FI  
 $DEST \leftarrow SRC \gg (TEMP * 8)$   
 $DEST[MAXVL-1:128] \leftarrow 0$ ;

**PSRLDQ(128-bit Legacy SSE version)**

$TEMP \leftarrow COUNT$   
 IF  $(TEMP > 15)$  THEN  $TEMP \leftarrow 16$ ; FI  
 $DEST \leftarrow DEST \gg (TEMP * 8)$   
 $DEST[MAXVL-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalents**

(V)PSRLDQ \_\_m128i \_mm\_srli\_si128 ( \_\_m128i a, int imm)  
 VPSRLDQ \_\_m256i \_mm256\_bsrl\_i\_epi128 ( \_\_m256i, const int)  
 VPSRLDQ \_\_m512i \_mm512\_bsrl\_i\_epi128 ( \_\_m512i, int)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 7.  
 EVEX-encoded instruction, see Exceptions Type E4NF.nb.

## PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF D1 /r <sup>1</sup> PSRLW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Shift words in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 OF D1 /r PSRLW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Shift words in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
NP OF 71 /2 ib <sup>1</sup> PSRLW <i>mm</i> , <i>imm8</i>	B	V/V	MMX	Shift words in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 OF 71 /2 ib PSRLW <i>xmm1</i> , <i>imm8</i>	B	V/V	SSE2	Shift words in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
NP OF D2 /r <sup>1</sup> PSRLD <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Shift doublewords in <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 OF D2 /r PSRLD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
NP OF 72 /2 ib <sup>1</sup> PSRLD <i>mm</i> , <i>imm8</i>	B	V/V	MMX	Shift doublewords in <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 OF 72 /2 ib PSRLD <i>xmm1</i> , <i>imm8</i>	B	V/V	SSE2	Shift doublewords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
NP OF D3 /r <sup>1</sup> PSRLQ <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Shift <i>mm</i> right by amount specified in <i>mm/m64</i> while shifting in 0s.
66 OF D3 /r PSRLQ <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Shift quadwords in <i>xmm1</i> right by amount specified in <i>xmm2/m128</i> while shifting in 0s.
NP OF 73 /2 ib <sup>1</sup> PSRLQ <i>mm</i> , <i>imm8</i>	B	V/V	MMX	Shift <i>mm</i> right by <i>imm8</i> while shifting in 0s.
66 OF 73 /2 ib PSRLQ <i>xmm1</i> , <i>imm8</i>	B	V/V	SSE2	Shift quadwords in <i>xmm1</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.OF.WIG D1 /r VPSRLW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.OF.WIG 71 /2 ib VPSRLW <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	D	V/V	AVX	Shift words in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.OF.WIG D2 /r VPSRLD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.OF.WIG 72 /2 ib VPSRLD <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	D	V/V	AVX	Shift doublewords in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.128.66.OF.WIG D3 /r VPSRLQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX	Shift quadwords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.128.66.OF.WIG 73 /2 ib VPSRLQ <i>xmm1</i> , <i>xmm2</i> , <i>imm8</i>	D	V/V	AVX	Shift quadwords in <i>xmm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.OF.WIG D1 /r VPSRLW <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	C	V/V	AVX2	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.OF.WIG 71 /2 ib VPSRLW <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	D	V/V	AVX2	Shift words in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.

VEX.NDS.256.66.0F.WIG D2 /r VPSRLD <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	C	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	D	V/V	AVX2	Shift doublewords in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.
VEX.NDS.256.66.0F.WIG D3 /r VPSRLQ <i>ymm1</i> , <i>ymm2</i> , <i>xmm3/m128</i>	C	V/V	AVX2	Shift quadwords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ <i>ymm1</i> , <i>ymm2</i> , <i>imm8</i>	D	V/V	AVX2	Shift quadwords in <i>ymm2</i> right by <i>imm8</i> while shifting in 0s.
EVEX.NDS.128.66.0F.WIG D1 /r VPSRLW <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512VL AVX512BW	Shift words in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG D1 /r VPSRLW <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512VL AVX512BW	Shift words in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG D1 /r VPSRLW <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512BW	Shift words in <i>zmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2/m128</i> , <i>imm8</i>	E	V/V	AVX512VL AVX512BW	Shift words in <i>xmm2/m128</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2/m256</i> , <i>imm8</i>	E	V/V	AVX512VL AVX512BW	Shift words in <i>ymm2/m256</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.512.66.0F.WIG 71 /2 ib VPSRLW <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2/m512</i> , <i>imm8</i>	E	V/V	AVX512BW	Shift words in <i>zmm2/m512</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W0 D2 /r VPSRLD <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512VL AVX512F	Shift doublewords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W0 D2 /r VPSRLD <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512VL AVX512F	Shift doublewords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W0 D2 /r VPSRLD <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512F	Shift doublewords in <i>zmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.128.66.0F.W0 72 /2 ib VPSRLD <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2/m128/m32bcst</i> , <i>imm8</i>	F	V/V	AVX512VL AVX512F	Shift doublewords in <i>xmm2/m128/m32bcst</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.256.66.0F.W0 72 /2 ib VPSRLD <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2/m256/m32bcst</i> , <i>imm8</i>	F	V/V	AVX512VL AVX512F	Shift doublewords in <i>ymm2/m256/m32bcst</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDD.512.66.0F.W0 72 /2 ib VPSRLD <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2/m512/m32bcst</i> , <i>imm8</i>	F	V/V	AVX512F	Shift doublewords in <i>zmm2/m512/m32bcst</i> right by <i>imm8</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W1 D3 /r VPSRLQ <i>xmm1</i> { <i>k1</i> } <i>z</i> , <i>xmm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512VL AVX512F	Shift quadwords in <i>xmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W1 D3 /r VPSRLQ <i>ymm1</i> { <i>k1</i> } <i>z</i> , <i>ymm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512VL AVX512F	Shift quadwords in <i>ymm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W1 D3 /r VPSRLQ <i>zmm1</i> { <i>k1</i> } <i>z</i> , <i>zmm2</i> , <i>xmm3/m128</i>	G	V/V	AVX512F	Shift quadwords in <i>zmm2</i> right by amount specified in <i>xmm3/m128</i> while shifting in 0s using writemask <i>k1</i> .

EVEX.NDD.128.66.0F.W1 73 /2 ib VPSRLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /2 ib VPSRLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

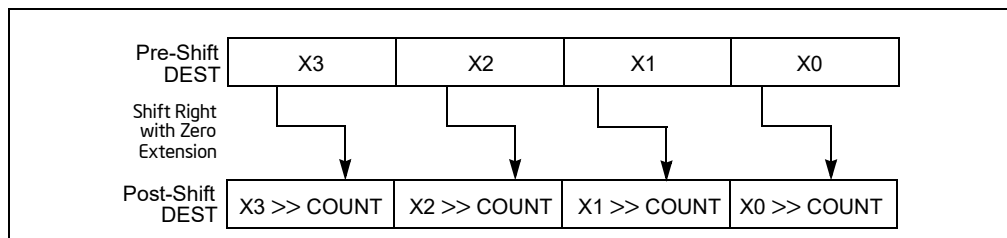
**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.



**Figure 4-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand**

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.



128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

## Operation

### PSRLW (with 64-bit operand)

```
IF (COUNT > 15)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
  FI;
```

### PSRLD (with 64-bit operand)

```
IF (COUNT > 31)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
  FI;
```

### PSRLQ (with 64-bit operand)

```
IF (COUNT > 63)
  THEN
    DEST[64:0] ← 0000000000000000H
  ELSE
    DEST ← ZeroExtend(DEST >> COUNT);
  FI;
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
  THEN
    DEST[31:0] ← 0
  ELSE
```

```

    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ← ZeroExtend(SRC[255:240] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ← ZeroExtend(SRC[127:112] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] ← ZeroExtend(SRC[255:224] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ← ZeroExtend(SRC[127:96] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] ← ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] ← ZeroExtend(SRC[255:192] >> COUNT);
FI;

```

```

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ← 00000000000000000000000000000000H
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ← ZeroExtend(SRC[127:64] >> COUNT);
FI;

```

**VPSRLW (EVEX versions, xmm/m128)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSRLW (EVEX versions, imm8)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_128b(SRC1[127:0], imm8)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[255:0], imm8)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1[511:256], imm8)

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSRLW (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0;

**VPSRLW (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS\_256b(SRC1, imm8)

DEST[MAXVL-1:256] ← 0;

**VPSRLW (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(SRC1, SRC2)

DEST[MAXVL-1:128] ← 0

**VPSRLW (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(SRC1, imm8)

DEST[MAXVL-1:128] ← 0

**PSRLW (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(DEST, SRC)

DEST[MAXVL-1:128] (Unmodified)

**PSRLW (xmm, imm8)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_WORDS(DEST, imm8)

DEST[MAXVL-1:128] (Unmodified)

**VPSRLD (EVEX versions, xmm/m128)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSRLD (EVEX versions, imm8)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+31:i] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS1(SRC1[31:0], imm8)

ELSE DEST[i+31:i] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS1(SRC1[i+31:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSRLD (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0;

**VPSRLD (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS\_256b(SRC1, imm8)

DEST[MAXVL-1:256] ← 0;

**VPSRLD (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS(SRC1, SRC2)

DEST[MAXVL-1:128] ← 0

**VPSRLD (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS(SRC1, imm8)

DEST[MAXVL-1:128] ← 0

**PSRLD (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS(DEST, SRC)

DEST[MAXVL-1:128] (Unmodified)

**PSRLD (xmm, imm8)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_DWORDS(DEST, imm8)

DEST[MAXVL-1:128] (Unmodified)

**VPSRLQ (EVEX versions, xmm/m128)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[511:256], SRC2)

IF VL = 128

TMP\_DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_128b(SRC1[127:0], SRC2)

FI;

IF VL = 256

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

FI;

IF VL = 512

TMP\_DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[255:0], SRC2)

TMP\_DEST[511:256] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1[511:256], SRC2)

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSRLQ (EVEX versions, imm8)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+63:i] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS1(SRC1[63:0], imm8)

ELSE DEST[i+63:i] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS1(SRC1[i+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0;

**VPSRLQ (ymm, imm8) - VEX.256 encoding**

DEST[255:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS\_256b(SRC1, imm8)

DEST[MAXVL-1:256] ← 0;

**VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(SRC1, SRC2)

DEST[MAXVL-1:128] ← 0

**VPSRLQ (xmm, imm8) - VEX.128 encoding**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(SRC1, imm8)

DEST[MAXVL-1:128] ← 0

**PSRLQ (xmm, xmm, xmm/m128)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(DEST, SRC)

DEST[MAXVL-1:128] (Unmodified)

**PSRLQ (xmm, imm8)**

DEST[127:0] ← LOGICAL\_RIGHT\_SHIFT\_QWORDS(DEST, imm8)

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalents**

VPSRLD \_\_m512i \_mm512\_srl\_epi32(\_\_m512i a, unsigned int imm);

VPSRLD \_\_m512i \_mm512\_mask\_srl\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, unsigned int imm);

VPSRLD \_\_m512i \_mm512\_maskz\_srl\_epi32(\_\_mmask16 k, \_\_m512i a, unsigned int imm);

VPSRLD \_\_m256i \_mm256\_mask\_srl\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, unsigned int imm);

VPSRLD \_\_m256i \_mm256\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m256i a, unsigned int imm);

VPSRLD \_\_m128i \_mm\_mask\_srl\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRLD \_\_m128i \_mm\_maskz\_srl\_epi32(\_\_mmask8 k, \_\_m128i a, unsigned int imm);

VPSRLD \_\_m512i \_mm512\_srl\_epi32(\_\_m512i a, \_\_m128i cnt);

VPSRLD \_\_m512i \_mm512\_mask\_srl\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);

VPSRLD \_\_m512i \_mm512\_maskz\_srl\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m128i cnt);

VPSRLD \_\_m256i \_mm256\_mask\_srl\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i cnt);

```

VPSRLD __m256i __mm256_maskz_srl_epi32( __mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m128i __mm_mask_srl_epi32( __m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLD __m128i __mm_maskz_srl_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i __mm512_srli_epi64( __m512i a, unsigned int imm);
VPSRLQ __m512i __mm512_mask_srli_epi64( __m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i __mm512_mask_srli_epi64( __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i __mm256_mask_srli_epi64( __m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i __mm256_maskz_srli_epi64( __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i __mm_mask_srli_epi64( __m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i __mm_maskz_srli_epi64( __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i __mm512_srl_epi64( __m512i a, __m128i cnt);
VPSRLQ __m512i __mm512_mask_srl_epi64( __m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m512i __mm512_mask_srl_epi64( __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m256i __mm256_mask_srl_epi64( __m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m256i __mm256_maskz_srl_epi64( __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m128i __mm_mask_srl_epi64( __m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m128i __mm_maskz_srl_epi64( __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m512i __mm512_srli_epi16( __m512i a, unsigned int imm);
VPSRLW __m512i __mm512_mask_srli_epi16( __m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m512i __mm512_maskz_srli_epi16( __mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m256i __mm256_mask_srlii_epi16( __m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m256i __mm256_maskz_srli_epi16( __mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m128i __mm_mask_srli_epi16( __m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m128i __mm_maskz_srli_epi16( __mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m512i __mm512_srl_epi16( __m512i a, __m128i cnt);
VPSRLW __m512i __mm512_mask_srl_epi16( __m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m512i __mm512_maskz_srl_epi16( __mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m256i __mm256_mask_srl_epi16( __m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSRLW __m256i __mm256_maskz_srl_epi16( __mmask8 k, __mmask16 a, __m128i cnt);
VPSRLW __m128i __mm_mask_srl_epi16( __m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m128i __mm_maskz_srl_epi16( __mmask8 k, __m128i a, __m128i cnt);
PSRLW: __m64 __mm_srli_pi16( __m64 m, int count)
PSRLW: __m64 __mm_srl_pi16( __m64 m, __m64 count)
(V)PSRLW: __m128i __mm_srli_epi16( __m128i m, int count)
(V)PSRLW: __m128i __mm_srl_epi16( __m128i m, __m128i count)
VPSRLW: __m256i __mm256_srli_epi16( __m256i m, int count)
VPSRLW: __m256i __mm256_srl_epi16( __m256i m, __m128i count)
PSRLD: __m64 __mm_srli_pi32( __m64 m, int count)
PSRLD: __m64 __mm_srl_pi32( __m64 m, __m64 count)
(V)PSRLD: __m128i __mm_srli_epi32( __m128i m, int count)
(V)PSRLD: __m128i __mm_srl_epi32( __m128i m, __m128i count)
VPSRLD: __m256i __mm256_srli_epi32( __m256i m, int count)
VPSRLD: __m256i __mm256_srl_epi32( __m256i m, __m128i count)
PSRLQ: __m64 __mm_srli_si64( __m64 m, int count)
PSRLQ: __m64 __mm_srl_si64( __m64 m, __m64 count)
(V)PSRLQ: __m128i __mm_srli_epi64( __m128i m, int count)
(V)PSRLQ: __m128i __mm_srl_epi64( __m128i m, __m128i count)
VPSRLQ: __m256i __mm256_srli_epi64( __m256i m, int count)
VPSRLQ: __m256i __mm256_srl_epi64( __m256i m, __m128i count)

```

## Flags Affected

None.



## Numeric Exceptions

None.

## Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding (A/C in the operand encoding table), see Exceptions Type 4.

Syntax with MI/VMI operand encoding (B/D in the operand encoding table), see Exceptions Type 7.

EVEX-encoded VPSRLW (E in the operand encoding table), see Exceptions Type E4NF.nb.

EVEX-encoded VPSRLD/Q:

Syntax with Mem128 tuple type (G in the operand encoding table), see Exceptions Type E4NF.nb.

Syntax with Full tuple type (F in the operand encoding table), see Exceptions Type E4.

## PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F8 /r <sup>1</sup> PSUBB <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Subtract packed byte integers in <i>mm/m64</i> from packed byte integers in <i>mm</i> .
66 0F F8 /r PSUBB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Subtract packed byte integers in <i>xmm2/m128</i> from packed byte integers in <i>xmm1</i> .
NP 0F F9 /r <sup>1</sup> PSUBW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Subtract packed word integers in <i>mm/m64</i> from packed word integers in <i>mm</i> .
66 0F F9 /r PSUBW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Subtract packed word integers in <i>xmm2/m128</i> from packed word integers in <i>xmm1</i> .
NP 0F FA /r <sup>1</sup> PSUBD <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Subtract packed doubleword integers in <i>mm/m64</i> from packed doubleword integers in <i>mm</i> .
66 0F FA /r PSUBD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Subtract packed doubleword integers in <i>xmm2/mem128</i> from packed doubleword integers in <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG F8 /r VPSUBB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Subtract packed byte integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG F9 /r VPSUBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Subtract packed word integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.128.66.0F.WIG FA /r VPSUBD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Subtract packed doubleword integers in <i>xmm3/m128</i> from <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG F8 /r VPSUBB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Subtract packed byte integers in <i>ymm3/m256</i> from <i>ymm2</i> .
VEX.NDS.256.66.0F.WIG F9 /r VPSUBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Subtract packed word integers in <i>ymm3/m256</i> from <i>ymm2</i> .
VEX.NDS.256.66.0F.WIG FA /r VPSUBD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Subtract packed doubleword integers in <i>ymm3/m256</i> from <i>ymm2</i> .
EVEX.NDS.128.66.0F.WIG F8 /r VPSUBB <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Subtract packed byte integers in <i>xmm3/m128</i> from <i>xmm2</i> and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG F8 /r VPSUBB <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Subtract packed byte integers in <i>ymm3/m256</i> from <i>ymm2</i> and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG F8 /r VPSUBB <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Subtract packed byte integers in <i>zmm3/m512</i> from <i>zmm2</i> and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.WIG F9 /r VPSUBW <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Subtract packed word integers in <i>xmm3/m128</i> from <i>xmm2</i> and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG F9 /r VPSUBW <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Subtract packed word integers in <i>ymm3/m256</i> from <i>ymm2</i> and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG F9 /r VPSUBW <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Subtract packed word integers in <i>zmm3/m512</i> from <i>zmm2</i> and store in <i>zmm1</i> using writemask <i>k1</i> .

EVEX.NDS.128.66.0F.W0 FA /r VPSUBD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in xmm3/m128/m32bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 FA /r VPSUBD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in ymm3/m256/m32bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 FA /r VPSUBD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Subtract packed doubleword integers in zmm3/m512/m32bcst from zmm2 and store in zmm1 using writemask k1

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: The second source operand is a ZMM/ymm/xmm register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/ymm/xmm registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/ymm/xmm register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/ymm/xmm registers. The destination is conditionally updated with writemask k1.

## Operation

### PSUBB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] – SRC[7:0];
(* Repeat subtract operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] – SRC[63:56];
```

### PSUBW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] – SRC[15:0];
(* Repeat subtract operation for 2nd and 3rd word *)
DEST[63:48] ← DEST[63:48] – SRC[63:48];
```

### PSUBD (with 64-bit operands)

```
DEST[31:0] ← DEST[31:0] – SRC[31:0];
DEST[63:32] ← DEST[63:32] – SRC[63:32];
```

### PSUBD (with 128-bit operands)

```
DEST[31:0] ← DEST[31:0] – SRC[31:0];
(* Repeat subtract operation for 2nd and 3rd doubleword *)
DEST[127:96] ← DEST[127:96] – SRC[127:96];
```

### VPSUBB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SRC1[i+7:i] - SRC2[i+7:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

### VPSUBW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC1[i+15:i] - SRC2[i+15:i]
  ELSE
    IF *merging-masking* ; merging-masking
```

```

        THEN *DEST[j+15:i] remains unchanged*
        ELSE *zeroing-masking*           ; zeroing-masking
        DEST[j+15:i] = 0
    FI
FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**VPSUBD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[j+31:i] ← SRC1[j+31:i] - SRC2[31:0]
            ELSE DEST[j+31:i] ← SRC1[j+31:i] - SRC2[j+31:i]
        FI;
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[j+31:i] remains unchanged*
            ELSE *zeroing-masking*     ; zeroing-masking
            DEST[j+31:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**VPSUBB (VEX.256 encoded version)**

```

DEST[7:0] ← SRC1[7:0]-SRC2[7:0]
DEST[15:8] ← SRC1[15:8]-SRC2[15:8]
DEST[23:16] ← SRC1[23:16]-SRC2[23:16]
DEST[31:24] ← SRC1[31:24]-SRC2[31:24]
DEST[39:32] ← SRC1[39:32]-SRC2[39:32]
DEST[47:40] ← SRC1[47:40]-SRC2[47:40]
DEST[55:48] ← SRC1[55:48]-SRC2[55:48]
DEST[63:56] ← SRC1[63:56]-SRC2[63:56]
DEST[71:64] ← SRC1[71:64]-SRC2[71:64]
DEST[79:72] ← SRC1[79:72]-SRC2[79:72]
DEST[87:80] ← SRC1[87:80]-SRC2[87:80]
DEST[95:88] ← SRC1[95:88]-SRC2[95:88]
DEST[103:96] ← SRC1[103:96]-SRC2[103:96]
DEST[111:104] ← SRC1[111:104]-SRC2[111:104]
DEST[119:112] ← SRC1[119:112]-SRC2[119:112]
DEST[127:120] ← SRC1[127:120]-SRC2[127:120]
DEST[135:128] ← SRC1[135:128]-SRC2[135:128]
DEST[143:136] ← SRC1[143:136]-SRC2[143:136]
DEST[151:144] ← SRC1[151:144]-SRC2[151:144]
DEST[159:152] ← SRC1[159:152]-SRC2[159:152]
DEST[167:160] ← SRC1[167:160]-SRC2[167:160]
DEST[175:168] ← SRC1[175:168]-SRC2[175:168]
DEST[183:176] ← SRC1[183:176]-SRC2[183:176]
DEST[191:184] ← SRC1[191:184]-SRC2[191:184]
DEST[199:192] ← SRC1[199:192]-SRC2[199:192]
DEST[207:200] ← SRC1[207:200]-SRC2[207:200]

```

DEST[215:208]  $\leftarrow$  SRC1[215:208]-SRC2[215:208]  
 DEST[223:216]  $\leftarrow$  SRC1[223:216]-SRC2[223:216]  
 DEST[231:224]  $\leftarrow$  SRC1[231:224]-SRC2[231:224]  
 DEST[239:232]  $\leftarrow$  SRC1[239:232]-SRC2[239:232]  
 DEST[247:240]  $\leftarrow$  SRC1[247:240]-SRC2[247:240]  
 DEST[255:248]  $\leftarrow$  SRC1[255:248]-SRC2[255:248]  
 DEST[MAXVL-1:256]  $\leftarrow$  0

#### VPSUBB (VEX.128 encoded version)

DEST[7:0]  $\leftarrow$  SRC1[7:0]-SRC2[7:0]  
 DEST[15:8]  $\leftarrow$  SRC1[15:8]-SRC2[15:8]  
 DEST[23:16]  $\leftarrow$  SRC1[23:16]-SRC2[23:16]  
 DEST[31:24]  $\leftarrow$  SRC1[31:24]-SRC2[31:24]  
 DEST[39:32]  $\leftarrow$  SRC1[39:32]-SRC2[39:32]  
 DEST[47:40]  $\leftarrow$  SRC1[47:40]-SRC2[47:40]  
 DEST[55:48]  $\leftarrow$  SRC1[55:48]-SRC2[55:48]  
 DEST[63:56]  $\leftarrow$  SRC1[63:56]-SRC2[63:56]  
 DEST[71:64]  $\leftarrow$  SRC1[71:64]-SRC2[71:64]  
 DEST[79:72]  $\leftarrow$  SRC1[79:72]-SRC2[79:72]  
 DEST[87:80]  $\leftarrow$  SRC1[87:80]-SRC2[87:80]  
 DEST[95:88]  $\leftarrow$  SRC1[95:88]-SRC2[95:88]  
 DEST[103:96]  $\leftarrow$  SRC1[103:96]-SRC2[103:96]  
 DEST[111:104]  $\leftarrow$  SRC1[111:104]-SRC2[111:104]  
 DEST[119:112]  $\leftarrow$  SRC1[119:112]-SRC2[119:112]  
 DEST[127:120]  $\leftarrow$  SRC1[127:120]-SRC2[127:120]  
 DEST[MAXVL-1:128]  $\leftarrow$  0

#### PSUBB (128-bit Legacy SSE version)

DEST[7:0]  $\leftarrow$  DEST[7:0]-SRC[7:0]  
 DEST[15:8]  $\leftarrow$  DEST[15:8]-SRC[15:8]  
 DEST[23:16]  $\leftarrow$  DEST[23:16]-SRC[23:16]  
 DEST[31:24]  $\leftarrow$  DEST[31:24]-SRC[31:24]  
 DEST[39:32]  $\leftarrow$  DEST[39:32]-SRC[39:32]  
 DEST[47:40]  $\leftarrow$  DEST[47:40]-SRC[47:40]  
 DEST[55:48]  $\leftarrow$  DEST[55:48]-SRC[55:48]  
 DEST[63:56]  $\leftarrow$  DEST[63:56]-SRC[63:56]  
 DEST[71:64]  $\leftarrow$  DEST[71:64]-SRC[71:64]  
 DEST[79:72]  $\leftarrow$  DEST[79:72]-SRC[79:72]  
 DEST[87:80]  $\leftarrow$  DEST[87:80]-SRC[87:80]  
 DEST[95:88]  $\leftarrow$  DEST[95:88]-SRC[95:88]  
 DEST[103:96]  $\leftarrow$  DEST[103:96]-SRC[103:96]  
 DEST[111:104]  $\leftarrow$  DEST[111:104]-SRC[111:104]  
 DEST[119:112]  $\leftarrow$  DEST[119:112]-SRC[119:112]  
 DEST[127:120]  $\leftarrow$  DEST[127:120]-SRC[127:120]  
 DEST[MAXVL-1:128] (Unmodified)

#### VPSUBW (VEX.256 encoded version)

DEST[15:0]  $\leftarrow$  SRC1[15:0]-SRC2[15:0]  
 DEST[31:16]  $\leftarrow$  SRC1[31:16]-SRC2[31:16]  
 DEST[47:32]  $\leftarrow$  SRC1[47:32]-SRC2[47:32]  
 DEST[63:48]  $\leftarrow$  SRC1[63:48]-SRC2[63:48]  
 DEST[79:64]  $\leftarrow$  SRC1[79:64]-SRC2[79:64]  
 DEST[95:80]  $\leftarrow$  SRC1[95:80]-SRC2[95:80]  
 DEST[111:96]  $\leftarrow$  SRC1[111:96]-SRC2[111:96]

DEST[127:112] ← SRC1[127:112]-SRC2[127:112]  
 DEST[143:128] ← SRC1[143:128]-SRC2[143:128]  
 DEST[159:144] ← SRC1[159:144]-SRC2[159:144]  
 DEST[175:160] ← SRC1[175:160]-SRC2[175:160]  
 DEST[191:176] ← SRC1[191:176]-SRC2[191:176]  
 DEST[207:192] ← SRC1[207:192]-SRC2[207:192]  
 DEST[223:208] ← SRC1[223:208]-SRC2[223:208]  
 DEST[239:224] ← SRC1[239:224]-SRC2[239:224]  
 DEST[255:240] ← SRC1[255:240]-SRC2[255:240]  
 DEST[MAXVL-1:256] ← 0

**VPSUBW (VEX.128 encoded version)**

DEST[15:0] ← SRC1[15:0]-SRC2[15:0]  
 DEST[31:16] ← SRC1[31:16]-SRC2[31:16]  
 DEST[47:32] ← SRC1[47:32]-SRC2[47:32]  
 DEST[63:48] ← SRC1[63:48]-SRC2[63:48]  
 DEST[79:64] ← SRC1[79:64]-SRC2[79:64]  
 DEST[95:80] ← SRC1[95:80]-SRC2[95:80]  
 DEST[111:96] ← SRC1[111:96]-SRC2[111:96]  
 DEST[127:112] ← SRC1[127:112]-SRC2[127:112]  
 DEST[MAXVL-1:128] ← 0

**PSUBW (128-bit Legacy SSE version)**

DEST[15:0] ← DEST[15:0]-SRC[15:0]  
 DEST[31:16] ← DEST[31:16]-SRC[31:16]  
 DEST[47:32] ← DEST[47:32]-SRC[47:32]  
 DEST[63:48] ← DEST[63:48]-SRC[63:48]  
 DEST[79:64] ← DEST[79:64]-SRC[79:64]  
 DEST[95:80] ← DEST[95:80]-SRC[95:80]  
 DEST[111:96] ← DEST[111:96]-SRC[111:96]  
 DEST[127:112] ← DEST[127:112]-SRC[127:112]  
 DEST[MAXVL-1:128] (Unmodified)

**VPSUBD (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0]-SRC2[31:0]  
 DEST[63:32] ← SRC1[63:32]-SRC2[63:32]  
 DEST[95:64] ← SRC1[95:64]-SRC2[95:64]  
 DEST[127:96] ← SRC1[127:96]-SRC2[127:96]  
 DEST[159:128] ← SRC1[159:128]-SRC2[159:128]  
 DEST[191:160] ← SRC1[191:160]-SRC2[191:160]  
 DEST[223:192] ← SRC1[223:192]-SRC2[223:192]  
 DEST[255:224] ← SRC1[255:224]-SRC2[255:224]  
 DEST[MAXVL-1:256] ← 0

**VPSUBD (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0]-SRC2[31:0]  
 DEST[63:32] ← SRC1[63:32]-SRC2[63:32]  
 DEST[95:64] ← SRC1[95:64]-SRC2[95:64]  
 DEST[127:96] ← SRC1[127:96]-SRC2[127:96]  
 DEST[MAXVL-1:128] ← 0

**PSUBD (128-bit Legacy SSE version)**

DEST[31:0] ← DEST[31:0]-SRC[31:0]  
 DEST[63:32] ← DEST[63:32]-SRC[63:32]

DEST[95:64] ← DEST[95:64]-SRC[95:64]  
 DEST[127:96] ← DEST[127:96]-SRC[127:96]  
 DEST[MAXVL-1:128] (Unmodified)

### Intel C/C++ Compiler Intrinsic Equivalents

VPSUBB \_\_m512i \_mm512\_sub\_epi8(\_\_m512i a, \_\_m512i b);  
 VPSUBB \_\_m512i \_mm512\_mask\_sub\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBB \_\_m512i \_mm512\_maskz\_sub\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBB \_\_m256i \_mm256\_mask\_sub\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBB \_\_m256i \_mm256\_maskz\_sub\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBB \_\_m128i \_mm\_mask\_sub\_epi8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBB \_\_m128i \_mm\_maskz\_sub\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBW \_\_m512i \_mm512\_sub\_epi16(\_\_m512i a, \_\_m512i b);  
 VPSUBW \_\_m512i \_mm512\_mask\_sub\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBW \_\_m512i \_mm512\_maskz\_sub\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBW \_\_m256i \_mm256\_mask\_sub\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBW \_\_m256i \_mm256\_maskz\_sub\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBW \_\_m128i \_mm\_mask\_sub\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBW \_\_m128i \_mm\_maskz\_sub\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBD \_\_m512i \_mm512\_sub\_epi32(\_\_m512i a, \_\_m512i b);  
 VPSUBD \_\_m512i \_mm512\_mask\_sub\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPSUBD \_\_m512i \_mm512\_maskz\_sub\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPSUBD \_\_m256i \_mm256\_mask\_sub\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPSUBD \_\_m256i \_mm256\_maskz\_sub\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPSUBD \_\_m128i \_mm\_mask\_sub\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBD \_\_m128i \_mm\_maskz\_sub\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PSUBB: \_\_m64 \_mm\_sub\_pi8(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBB: \_\_m128i \_mm\_sub\_epi8 (\_\_m128i a, \_\_m128i b)  
 VPSUBB: \_\_m256i \_mm256\_sub\_epi8 (\_\_m256i a, \_\_m256i b)  
 PSUBW: \_\_m64 \_mm\_sub\_pi16(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBW: \_\_m128i \_mm\_sub\_epi16 (\_\_m128i a, \_\_m128i b)  
 VPSUBW: \_\_m256i \_mm256\_sub\_epi16 (\_\_m256i a, \_\_m256i b)  
 PSUBD: \_\_m64 \_mm\_sub\_pi32(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBD: \_\_m128i \_mm\_sub\_epi32 (\_\_m128i a, \_\_m128i b)  
 VPSUBD: \_\_m256i \_mm256\_sub\_epi32 (\_\_m256i a, \_\_m256i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBD, see Exceptions Type E4.

EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb.



## PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FB /r <sup>1</sup> PSUBQ <i>mm1</i> , <i>mm2</i> /m64	A	V/V	SSE2	Subtract quadword integer in <i>mm1</i> from <i>mm2</i> /m64.
66 0F FB /r PSUBQ <i>xmm1</i> , <i>xmm2</i> /m128	A	V/V	SSE2	Subtract packed quadword integers in <i>xmm1</i> from <i>xmm2</i> /m128.
VEX.NDS.128.66.0F.WIG FB/r VPSUBQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3</i> /m128	B	V/V	AVX	Subtract packed quadword integers in <i>xmm3</i> /m128 from <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG FB /r VPSUBQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3</i> /m256	B	V/V	AVX2	Subtract packed quadword integers in <i>ymm3</i> /m256 from <i>ymm2</i> .
EVEX.NDS.128.66.0F.W1 FB /r VPSUBQ <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3</i> /m128/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed quadword integers in <i>xmm3</i> /m128/m64bcst from <i>xmm2</i> and store in <i>xmm1</i> using writemask k1.
EVEX.NDS.256.66.0F.W1 FB /r VPSUBQ <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3</i> /m256/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed quadword integers in <i>ymm3</i> /m256/m64bcst from <i>ymm2</i> and store in <i>ymm1</i> using writemask k1.
EVEX.NDS.512.66.0F.W1 FB/r VPSUBQ <i>zmm1</i> {k1}{z}, <i>zmm2</i> , <i>zmm3</i> /m512/m64bcst	C	V/V	AVX512F	Subtract packed quadword integers in <i>zmm3</i> /m512/m64bcst from <i>zmm2</i> and store in <i>zmm1</i> using writemask k1.

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: The second source operand is a ZMM/ymm/xmm register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/ymm/xmm registers. The destination is conditionally updated with writemask k1.

## Operation

### PSUBQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] – SRC[63:0];
```

### PSUBQ (with 128-Bit operands)

```
DEST[63:0] ← DEST[63:0] – SRC[63:0];
DEST[127:64] ← DEST[127:64] – SRC[127:64];
```

### VPSUBQ (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[MAXVL-1:128] ← 0
```

### VPSUBQ (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[191:128] ← SRC1[191:128]-SRC2[191:128]
DEST[255:192] ← SRC1[255:192]-SRC2[255:192]
DEST[MAXVL-1:256] ← 0
```

### VPSUBQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[63:0]

      ELSE DEST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i]

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPSUBQ __m512i _mm512_sub_epi64(__m512i a, __m512i b);
VPSUBQ __m512i _mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m512i _mm512_maskz_sub_epi64(__mmask8 k, __m512i a, __m512i b);
VPSUBQ __m256i _mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBQ __m256i _mm256_maskz_sub_epi64(__mmask8 k, __m256i a, __m256i b);
VPSUBQ __m128i _mm_mask_sub_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBQ __m128i _mm_maskz_sub_epi64(__mmask8 k, __m128i a, __m128i b);
PSUBQ: __m64 _mm_sub_si64(__m64 m1, __m64 m2)
(V)PSUBQ: __m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBQ: __m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

```

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBQ, see Exceptions Type E4.

## PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E8 /r <sup>1</sup> PSUBSB <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Subtract signed packed bytes in <i>mm/m64</i> from signed packed bytes in <i>mm</i> and saturate results.
66 0F E8 /r PSUBSB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Subtract packed signed byte integers in <i>xmm2/m128</i> from packed signed byte integers in <i>xmm1</i> and saturate results.
NP 0F E9 /r <sup>1</sup> PSUBSW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Subtract signed packed words in <i>mm/m64</i> from signed packed words in <i>mm</i> and saturate results.
66 0F E9 /r PSUBSW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Subtract packed signed word integers in <i>xmm2/m128</i> from packed signed word integers in <i>xmm1</i> and saturate results.
VEX.NDS.128.66.0F.WIG E8 /r VPSUBSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Subtract packed signed byte integers in <i>xmm3/m128</i> from packed signed byte integers in <i>xmm2</i> and saturate results.
VEX.NDS.128.66.0F.WIG E9 /r VPSUBSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Subtract packed signed word integers in <i>xmm3/m128</i> from packed signed word integers in <i>xmm2</i> and saturate results.
VEX.NDS.256.66.0F.WIG E8 /r VPSUBSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Subtract packed signed byte integers in <i>ymm3/m256</i> from packed signed byte integers in <i>ymm2</i> and saturate results.
VEX.NDS.256.66.0F.WIG E9 /r VPSUBSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Subtract packed signed word integers in <i>ymm3/m256</i> from packed signed word integers in <i>ymm2</i> and saturate results.
EVEX.NDS.128.66.0F.WIG E8 /r VPSUBSB <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in <i>xmm3/m128</i> from packed signed byte integers in <i>xmm2</i> and saturate results and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG E8 /r VPSUBSB <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in <i>ymm3/m256</i> from packed signed byte integers in <i>ymm2</i> and saturate results and store in <i>ymm1</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.WIG E8 /r VPSUBSB <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Subtract packed signed byte integers in <i>zmm3/m512</i> from packed signed byte integers in <i>zmm2</i> and saturate results and store in <i>zmm1</i> using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.WIG E9 /r VPSUBSW <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in <i>xmm3/m128</i> from packed signed word integers in <i>xmm2</i> and saturate results and store in <i>xmm1</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.WIG E9 /r VPSUBSW <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in <i>ymm3/m256</i> from packed signed word integers in <i>ymm2</i> and saturate results and store in <i>ymm1</i> using writemask <i>k1</i> .

EVEX.NDS.512.66.0F.WIG E9 /r VPSUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1.
---	---	-----	----------	---

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or a 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation****PSUBSB (with 64-bit operands)**

DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC (7:0));

(\* Repeat subtract operation for 2nd through 7th bytes \*)

DEST[63:56] ← SaturateToSignedByte (DEST[63:56] – SRC[63:56] );

**PSUBSW (with 64-bit operands)**

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0] );
(* Repeat subtract operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord (DEST[63:48] – SRC[63:48] );
```

**VPSUBSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8;
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+7:i] ← 0;
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

**VPSUBSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0;
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

**VPSUBSB (VEX.256 encoded version)**

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] ← 0;
```

**VPSUBSB (VEX.128 encoded version)**

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] ← 0;
```

**PSUBSB (128-bit Legacy SSE Version)**

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified);
```

**VPSUBSW (VEX.256 encoded version)**

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 15th words \*)  
 DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);  
 DEST[MAXVL-1:256] ← 0;

**VPSUBSW (VEX.128 encoded version)**

DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);  
 DEST[MAXVL-1:128] ← 0;

**PSUBSW (128-bit Legacy SSE Version)**

DEST[15:0] ← SaturateToSignedWord (DEST[15:0] - SRC[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToSignedWord (DEST[127:112] - SRC[127:112]);  
 DEST[MAXVL-1:128] (Unmodified);

**Intel C/C++ Compiler Intrinsic Equivalents**

VPSUBSB \_\_m512i\_mm512\_subs\_epi8(\_\_m512i a, \_\_m512i b);  
 VPSUBSB \_\_m512i\_mm512\_mask\_subs\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSB \_\_m512i\_mm512\_maskz\_subs\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSB \_\_m256i\_mm256\_mask\_subs\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSB \_\_m256i\_mm256\_maskz\_subs\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSB \_\_m128i\_mm\_mask\_subs\_epi8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBSB \_\_m128i\_mm\_maskz\_subs\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBSW \_\_m512i\_mm512\_subs\_epi16(\_\_m512i a, \_\_m512i b);  
 VPSUBSW \_\_m512i\_mm512\_mask\_subs\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSW \_\_m512i\_mm512\_maskz\_subs\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBSW \_\_m256i\_mm256\_mask\_subs\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSW \_\_m256i\_mm256\_maskz\_subs\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBSW \_\_m128i\_mm\_mask\_subs\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBSW \_\_m128i\_mm\_maskz\_subs\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PSUBSB: \_\_m64\_mm\_subs\_pi8(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBSB: \_\_m128i\_mm\_subs\_epi8(\_\_m128i m1, \_\_m128i m2)  
 VPSUBSB: \_\_m256i\_mm256\_subs\_epi8(\_\_m256i m1, \_\_m256i m2)  
 PSUBSW: \_\_m64\_mm\_subs\_pi16(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBSW: \_\_m128i\_mm\_subs\_epi16(\_\_m128i m1, \_\_m128i m2)  
 VPSUBSW: \_\_m256i\_mm256\_subs\_epi16(\_\_m256i m1, \_\_m256i m2)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

## PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF D8 /r <sup>1</sup> PSUBUSB <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Subtract unsigned packed bytes in <i>mm/m64</i> from unsigned packed bytes in <i>mm</i> and saturate result.
66 OF D8 /r PSUBUSB <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Subtract packed unsigned byte integers in <i>xmm2/m128</i> from packed unsigned byte integers in <i>xmm1</i> and saturate result.
NP OF D9 /r <sup>1</sup> PSUBUSW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Subtract unsigned packed words in <i>mm/m64</i> from unsigned packed words in <i>mm</i> and saturate result.
66 OF D9 /r PSUBUSW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Subtract packed unsigned word integers in <i>xmm2/m128</i> from packed unsigned word integers in <i>xmm1</i> and saturate result.
VEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Subtract packed unsigned byte integers in <i>xmm3/m128</i> from packed unsigned byte integers in <i>xmm2</i> and saturate result.
VEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Subtract packed unsigned word integers in <i>xmm3/m128</i> from packed unsigned word integers in <i>xmm2</i> and saturate result.
VEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Subtract packed unsigned byte integers in <i>ymm3/m256</i> from packed unsigned byte integers in <i>ymm2</i> and saturate result.
VEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Subtract packed unsigned word integers in <i>ymm3/m256</i> from packed unsigned word integers in <i>ymm2</i> and saturate result.
EVEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in <i>xmm3/m128</i> from packed unsigned byte integers in <i>xmm2</i> , saturate results and store in <i>xmm1</i> using writemask k1.
EVEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in <i>ymm3/m256</i> from packed unsigned byte integers in <i>ymm2</i> , saturate results and store in <i>ymm1</i> using writemask k1.
EVEX.NDS.512.66.0F.WIG D8 /r VPSUBUSB <i>zmm1</i> {k1}{z}, <i>zmm2</i> , <i>zmm3/m512</i>	C	V/V	AVX512BW	Subtract packed unsigned byte integers in <i>zmm3/m512</i> from packed unsigned byte integers in <i>zmm2</i> , saturate results and store in <i>zmm1</i> using writemask k1.
EVEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in <i>xmm3/m128</i> from packed unsigned word integers in <i>xmm2</i> and saturate results and store in <i>xmm1</i> using writemask k1.
EVEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW <i>ymm1</i> {k1}{z}, <i>ymm2</i> , <i>ymm3/m256</i>	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in <i>ymm3/m256</i> from packed unsigned word integers in <i>ymm2</i> , saturate results and store in <i>ymm1</i> using writemask k1.



EVEX.NDS.512.66.0F.WIG D9 /r VPSUBUSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.
--	---	-----	----------	--

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

**Operation****PSUBUSB (with 64-bit operands)**

DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC (7:0));

(\* Repeat add operation for 2nd through 7th bytes \*)

DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] – SRC[63:56];

**PSUBUSW (with 64-bit operands)**

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] – SRC[63:48] );
```

**VPSUBUSB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8;
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] - SRC2[i+7:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] ← 0;
      FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

**VPSUBUSW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16;
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] - SRC2[i+15:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] ← 0;
      FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

**VPSUBUSB (VEX.256 encoded version)**

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] ← SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] ← 0;
```

**VPSUBUSW (VEX.128 encoded version)**

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] ← 0
```

**PSUBUSB (128-bit Legacy SSE Version)**

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified)
```

**VPSUBUSB (VEX.256 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 15th words \*)  
 DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);  
 DEST[MAXVL-1:256] ← 0;

**VPSUBUSB (VEX.128 encoded version)**

DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);  
 DEST[MAXVL-1:128] ← 0

**PSUBUSB (128-bit Legacy SSE Version)**

DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);  
 (\* Repeat subtract operation for 2nd through 7th words \*)  
 DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);  
 DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalents**

VPSUBUSB \_\_m512i \_\_mm512\_subs\_epu8(\_\_m512i a, \_\_m512i b);  
 VPSUBUSB \_\_m512i \_\_mm512\_mask\_subs\_epu8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSB \_\_m512i \_\_mm512\_maskz\_subs\_epu8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSB \_\_m256i \_\_mm256\_mask\_subs\_epu8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSB \_\_m256i \_\_mm256\_maskz\_subs\_epu8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSB \_\_m128i \_\_mm\_mask\_subs\_epu8(\_\_m128i s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBUSB \_\_m128i \_\_mm\_maskz\_subs\_epu8(\_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPSUBUSBW \_\_m512i \_\_mm512\_subs\_epu16(\_\_m512i a, \_\_m512i b);  
 VPSUBUSBW \_\_m512i \_\_mm512\_mask\_subs\_epu16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSBW \_\_m512i \_\_mm512\_maskz\_subs\_epu16(\_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPSUBUSBW \_\_m256i \_\_mm256\_mask\_subs\_epu16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSBW \_\_m256i \_\_mm256\_maskz\_subs\_epu16(\_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPSUBUSBW \_\_m128i \_\_mm\_mask\_subs\_epu16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPSUBUSBW \_\_m128i \_\_mm\_maskz\_subs\_epu16(\_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PSUBUSB: \_\_m64 \_\_mm\_subs\_pu8(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBUSB: \_\_m128i \_\_mm\_subs\_epu8(\_\_m128i m1, \_\_m128i m2)  
 VPSUBUSB: \_\_m256i \_\_mm256\_subs\_epu8(\_\_m256i m1, \_\_m256i m2)  
 PSUBUSBW: \_\_m64 \_\_mm\_subs\_pu16(\_\_m64 m1, \_\_m64 m2)  
 (V)PSUBUSBW: \_\_m128i \_\_mm\_subs\_epu16(\_\_m128i m1, \_\_m128i m2)  
 VPSUBUSBW: \_\_m256i \_\_mm256\_subs\_epu16(\_\_m256i m1, \_\_m256i m2)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## PTEST- Logical Compare

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 17 /r PTEST <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE4_1	Set ZF if <i>xmm2/m128</i> AND <i>xmm1</i> result is all 0s. Set CF if <i>xmm2/m128</i> AND NOT <i>xmm1</i> result is all 0s.
VEX.128.66.0F38.WIG 17 /r VPTEST <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.
VEX.256.66.0F38.WIG 17 /r VPTEST <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

### Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

The first source register is specified by the ModR/M *reg* field.

128-bit versions: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

#### (V)PTEST (128-bit version)

IF (SRC[127:0] BITWISE AND DEST[127:0] = 0)

THEN ZF ← 1;

ELSE ZF ← 0;

IF (SRC[127:0] BITWISE AND NOT DEST[127:0] = 0)

THEN CF ← 1;

ELSE CF ← 0;

DEST (unmodified)

AF ← OF ← PF ← SF ← 0;

#### VPTEST (VEX.256 encoded version)

IF (SRC[255:0] BITWISE AND DEST[255:0] = 0) THEN ZF ← 1;

ELSE ZF ← 0;

IF (SRC[255:0] BITWISE AND NOT DEST[255:0] = 0) THEN CF ← 1;

ELSE CF ← 0;

DEST (unmodified)

AF ← OF ← PF ← SF ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent****PTEST**

```
int _mm_testz_si128 (__m128i s1, __m128i s2);
int _mm_testc_si128 (__m128i s1, __m128i s2);
int _mm_testnzc_si128 (__m128i s1, __m128i s2);
```

**VPTEST**

```
int _mm256_testz_si256 (__m256i s1, __m256i s2);
int _mm256_testc_si256 (__m256i s1, __m256i s2);
int _mm256_testnzc_si256 (__m256i s1, __m256i s2);
int _mm_testz_si128 (__m128i s1, __m128i s2);
int _mm_testc_si128 (__m128i s1, __m128i s2);
int _mm_testnzc_si128 (__m128i s1, __m128i s2);
```

**Flags Affected**

The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 4; additionally

#UD                      If VEX.vvvv ≠ 1111B.

## PTWRITE - Write Data to a Processor Trace Packet

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 REX.W OF AE /4 PTWRITE <i>r64/m64</i>	RM	V/N.E		Reads the data from r64/m64 to encode into a PTW packet if dependencies are met (see details below).
F3 OF AE /4 PTWRITE <i>r32/m32</i>	RM	V/V		Reads the data from r32/m32 to encode into a PTW packet if dependencies are met (see details below).

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:rm (r)	NA	NA	NA

### Description

This instruction reads data in the source operand and sends it to the Intel Processor Trace hardware to be encoded in a PTW packet if TriggerEn, ContextEn, FilterEn, and PTWEn are all set to 1. For more details on these values, see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, Section 35.2.3, "Power Event Tracing". The size of data is 64-bit if using REX.W in 64-bit mode, otherwise 32-bits of data are copied from the source operand.

Note: The instruction will #UD if prefix 66H is used.

### Operation

IF (IA32\_RTIT\_STATUS.TriggerEn & IA32\_RTIT\_STATUS.ContextEn & IA32\_RTIT\_STATUS.FilterEn & IA32\_RTIT\_CTL.PTWEn) = 1

PTW.PayloadBytes ← Encoded payload size;

PTW.IP ← IA32\_RTIT\_CTL.FUPonPTW

IF IA32\_RTIT\_CTL.FUPonPTW = 1

Insert FUP packet with IP of PTWRITE;

FI;

FI;

### Flags Affected

None.

### Other Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWWRITE [Bit 4] = 0. If LOCK prefix is used. If 66H prefix is used.

**Real-Address Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0. If LOCK prefix is used. If 66H prefix is used.

**Virtual 8086 Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0. If LOCK prefix is used. If 66H prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0. If LOCK prefix is used. If 66H prefix is used.

## PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 68 /r <sup>1</sup> PUNPCKHBW <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Unpack and interleave high-order bytes from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 OF 68 /r PUNPCKHBW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Unpack and interleave high-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
NP OF 69 /r <sup>1</sup> PUNPCKHWD <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Unpack and interleave high-order words from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 OF 69 /r PUNPCKHWD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Unpack and interleave high-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
NP OF 6A /r <sup>1</sup> PUNPCKHDQ <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Unpack and interleave high-order doublewords from <i>mm</i> and <i>mm/m64</i> into <i>mm</i> .
66 OF 6A /r PUNPCKHDQ <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Unpack and interleave high-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 OF 6D /r PUNPCKHQDQ <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Unpack and interleave high-order quadwords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 68/r VPUNPCKHBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Interleave high-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 69/r VPUNPCKHWD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Interleave high-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 6A/r VPUNPCKHDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Interleave high-order doublewords from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 6D/r VPUNPCKHQDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Interleave high-order quadword from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register.
VEX.NDS.256.66.OF.WIG 68 /r VPUNPCKHBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Interleave high-order bytes from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 69 /r VPUNPCKHWD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Interleave high-order words from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 6A /r VPUNPCKHDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Interleave high-order doublewords from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 6D /r VPUNPCKHQDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Interleave high-order quadword from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
EVEX.NDS.128.66.OF.WIG 68 /r VPUNPCKHBW <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Interleave high-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register using k1 write mask.
EVEX.NDS.128.66.OF.WIG 69 /r VPUNPCKHWD <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Interleave high-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register using k1 write mask.
EVEX.NDS.128.66.OF.WO 6A /r VPUNPCKHDQ <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	D	V/V	AVX512VL AVX512F	Interleave high-order doublewords from <i>xmm2</i> and <i>xmm3/m128/m32bcst</i> into <i>xmm1</i> register using k1 write mask.
EVEX.NDS.128.66.OF.W1 6D /r VPUNPCKHQDQ <i>xmm1</i> {k1}{z}, <i>xmm2</i> , <i>xmm3/m128/m64bcst</i>	D	V/V	AVX512VL AVX512F	Interleave high-order quadword from <i>xmm2</i> and <i>xmm3/m128/m64bcst</i> into <i>xmm1</i> register using k1 write mask.



EVEX.NDS.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.W0 6A /r VPUNPCKHDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Interleave high-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.W1 6D /r VPUNPCKHQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Interleave high-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register using k1 write mask.
EVEX.NDS.512.66.0F.WIG 68/r VPUNPCKHBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave high-order bytes from zmm2 and zmm3/m512 into zmm1 register.
EVEX.NDS.512.66.0F.WIG 69/r VPUNPCKHWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave high-order words from zmm2 and zmm3/m512 into zmm1 register.
EVEX.NDS.512.66.0F.W0 6A /r VPUNPCKHDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Interleave high-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register using k1 write mask.
EVEX.NDS.512.66.0F.W1 6D /r VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F	Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask.

**NOTES:**

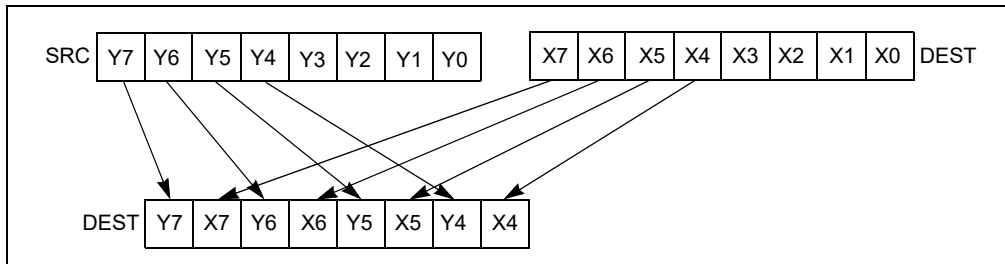
1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

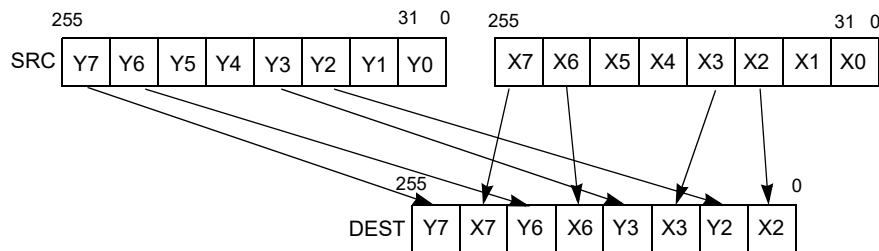
Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.



**Figure 4-20. PUNPCKHBW Instruction Operation Using 64-bit Operands**



**Figure 4-21. 256-bit VPUNPCKHDQ Instruction Operation**

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE versions 64-bit operand:** The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

**128-bit Legacy SSE versions:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded versions:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**VEX.256 encoded version:** The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

EVEX encoded VPUNPCKHDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

## Operation

### PUNPCKHBW instruction with 64-bit operands:

```
DEST[7:0] ← DEST[39:32];
DEST[15:8] ← SRC[39:32];
DEST[23:16] ← DEST[47:40];
DEST[31:24] ← SRC[47:40];
DEST[39:32] ← DEST[55:48];
DEST[47:40] ← SRC[55:48];
DEST[55:48] ← DEST[63:56];
DEST[63:56] ← SRC[63:56];
```

### PUNPCKHW instruction with 64-bit operands:

```
DEST[15:0] ← DEST[47:32];
DEST[31:16] ← SRC[47:32];
DEST[47:32] ← DEST[63:48];
DEST[63:48] ← SRC[63:48];
```

### PUNPCKHDQ instruction with 64-bit operands:

```
DEST[31:0] ← DEST[63:32];
DEST[63:32] ← SRC[63:32];
```

INTERLEAVE\_HIGH\_BYTES\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1[255:0], SRC[255:0])

TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_HIGH\_BYTES\_256b (SRC1, SRC2)

```
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]
DEST[135:128] ← SRC1[199:192]
DEST[143:136] ← SRC2[199:192]
DEST[151:144] ← SRC1[207:200]
DEST[159:152] ← SRC2[207:200]
```

DEST[167:160] ← SRC1[215:208]  
 DEST[175:168] ← SRC2[215:208]  
 DEST[183:176] ← SRC1[223:216]  
 DEST[191:184] ← SRC2[223:216]  
 DEST[199:192] ← SRC1[231:224]  
 DEST[207:200] ← SRC2[231:224]  
 DEST[215:208] ← SRC1[239:232]  
 DEST[223:216] ← SRC2[239:232]  
 DEST[231:224] ← SRC1[247:240]  
 DEST[239:232] ← SRC2[247:240]  
 DEST[247:240] ← SRC1[255:248]  
 DEST[255:248] ← SRC2[255:248]

INTERLEAVE\_HIGH\_BYTES (SRC1, SRC2)

DEST[7:0] ← SRC1[71:64]  
 DEST[15:8] ← SRC2[71:64]  
 DEST[23:16] ← SRC1[79:72]  
 DEST[31:24] ← SRC2[79:72]  
 DEST[39:32] ← SRC1[87:80]  
 DEST[47:40] ← SRC2[87:80]  
 DEST[55:48] ← SRC1[95:88]  
 DEST[63:56] ← SRC2[95:88]  
 DEST[71:64] ← SRC1[103:96]  
 DEST[79:72] ← SRC2[103:96]  
 DEST[87:80] ← SRC1[111:104]  
 DEST[95:88] ← SRC2[111:104]  
 DEST[103:96] ← SRC1[119:112]  
 DEST[111:104] ← SRC2[119:112]  
 DEST[119:112] ← SRC1[127:120]  
 DEST[127:120] ← SRC2[127:120]

INTERLEAVE\_HIGH\_WORDS\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1[255:0], SRC[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_HIGH\_WORDS\_256b(SRC1, SRC2)

DEST[15:0] ← SRC1[79:64]  
 DEST[31:16] ← SRC2[79:64]  
 DEST[47:32] ← SRC1[95:80]  
 DEST[63:48] ← SRC2[95:80]  
 DEST[79:64] ← SRC1[111:96]  
 DEST[95:80] ← SRC2[111:96]  
 DEST[111:96] ← SRC1[127:112]  
 DEST[127:112] ← SRC2[127:112]  
 DEST[143:128] ← SRC1[207:192]  
 DEST[159:144] ← SRC2[207:192]  
 DEST[175:160] ← SRC1[223:208]  
 DEST[191:176] ← SRC2[223:208]  
 DEST[207:192] ← SRC1[239:224]  
 DEST[223:208] ← SRC2[239:224]  
 DEST[239:224] ← SRC1[255:240]  
 DEST[255:240] ← SRC2[255:240]

INTERLEAVE\_HIGH\_WORDS (SRC1, SRC2)

DEST[15:0] ← SRC1[79:64]  
 DEST[31:16] ← SRC2[79:64]  
 DEST[47:32] ← SRC1[95:80]  
 DEST[63:48] ← SRC2[95:80]  
 DEST[79:64] ← SRC1[111:96]  
 DEST[95:80] ← SRC2[111:96]  
 DEST[111:96] ← SRC1[127:112]  
 DEST[127:112] ← SRC2[127:112]

INTERLEAVE\_HIGH\_DWORDS\_512b(SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_HIGH\_DWORDS\_256b(SRC1, SRC2)  
 DEST[31:0] ← SRC1[95:64]  
 DEST[63:32] ← SRC2[95:64]  
 DEST[95:64] ← SRC1[127:96]  
 DEST[127:96] ← SRC2[127:96]  
 DEST[159:128] ← SRC1[223:192]  
 DEST[191:160] ← SRC2[223:192]  
 DEST[223:192] ← SRC1[255:224]  
 DEST[255:224] ← SRC2[255:224]

INTERLEAVE\_HIGH\_DWORDS(SRC1, SRC2)  
 DEST[31:0] ← SRC1[95:64]  
 DEST[63:32] ← SRC2[95:64]  
 DEST[95:64] ← SRC1[127:96]  
 DEST[127:96] ← SRC2[127:96]

INTERLEAVE\_HIGH\_QWORDS\_512b(SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1, SRC2)  
 DEST[63:0] ← SRC1[127:64]  
 DEST[127:64] ← SRC2[127:64]  
 DEST[191:128] ← SRC1[255:192]  
 DEST[255:192] ← SRC2[255:192]

INTERLEAVE\_HIGH\_QWORDS(SRC1, SRC2)  
 DEST[63:0] ← SRC1[127:64]  
 DEST[127:64] ← SRC2[127:64]

#### **PUNPCKHBW (128-bit Legacy SSE Version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_BYTES(DEST, SRC)  
 DEST[255:127] (Unmodified)

#### **VPUNPCKHBW (VEX.128 encoded version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_BYTES(SRC1, SRC2)  
 DEST[MAXVL-1:127] ← 0

#### **VPUNPCKHBW (VEX.256 encoded version)**

DEST[255:0] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1, SRC2)  
 DEST[MAXVL-1:256] ← 0

**VPUNPCKHBW (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_BYTES\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_BYTES\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TMP\_DEST[i+7:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**PUNPCKHWD (128-bit Legacy SSE Version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_WORDS(DEST, SRC)

DEST[255:127] (Unmodified)

**VPUNPCKHWD (VEX.128 encoded version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_WORDS(SRC1, SRC2)

DEST[MAXVL-1:127] ← 0

**VPUNPCKHWD (VEX.256 encoded version)**

DEST[255:0] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0

**VPUNPCKHWD (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_WORDS\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_WORDS\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

```

    THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**PUNPCKHDQ (128-bit Legacy SSE Version)**

```

DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)

```

**VPUNPCKHDQ (VEX.128 encoded version)**

```

DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:127] ← 0

```

**VPUNPCKHDQ (VEX.256 encoded version)**

```

DEST[255:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

```

**VPUNPCKHDQ (EVEX.512 encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 128
  TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
  TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
  TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

```

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR

```

DEST[MAXVL-1:VL] ← 0

#### **PUNPCKHQDQ (128-bit Legacy SSE Version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_QWORDS(DEST, SRC)

DEST[MAXVL-1:128] (Unmodified)

#### **VPUNPCKHQDQ (VEX.128 encoded version)**

DEST[127:0] ← INTERLEAVE\_HIGH\_QWORDS(SRC1, SRC2)

DEST[MAXVL-1:128] ← 0

#### **VPUNPCKHQDQ (VEX.256 encoded version)**

DEST[255:0] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0

#### **VPUNPCKHQDQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN TMP\_SRC2[i+63:i] ← SRC2[63:0]

        ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i]

    FI;

ENDFOR;

IF VL = 128

    TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_QWORDS(SRC1[VL-1:0], TMP\_SRC2[VL-1:0])

FI;

IF VL = 256

    TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_QWORDS\_256b(SRC1[VL-1:0], TMP\_SRC2[VL-1:0])

FI;

IF VL = 512

    TMP\_DEST[VL-1:0] ← INTERLEAVE\_HIGH\_QWORDS\_512b(SRC1[VL-1:0], TMP\_SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

#### **Intel C/C++ Compiler Intrinsic Equivalents**

VPUNPCKHBW \_\_m512i \_mm512\_unpackhi\_epi8(\_\_m512i a, \_\_m512i b);

VPUNPCKHBW \_\_m512i \_mm512\_mask\_unpackhi\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPUNPCKHBW \_\_m512i \_mm512\_maskz\_unpackhi\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPUNPCKHBW \_\_m256i \_mm256\_mask\_unpackhi\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPUNPCKHBW \_\_m256i \_mm256\_maskz\_unpackhi\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPUNPCKHBW \_\_m128i \_mm\_mask\_unpackhi\_epi8(v s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);



VPUNPCKHBW \_\_m128i \_\_mm\_maskz\_unpackhi\_epi8( \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKHWD \_\_m512i \_\_mm512\_unpackhi\_epi16( \_\_m512i a, \_\_m512i b);  
 VPUNPCKHWD \_\_m512i \_\_mm512\_mask\_unpackhi\_epi16( \_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHWD \_\_m512i \_\_mm512\_maskz\_unpackhi\_epi16( \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHWD \_\_m256i \_\_mm256\_mask\_unpackhi\_epi16( \_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKHWD \_\_m256i \_\_mm256\_maskz\_unpackhi\_epi16( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKHWD \_\_m128i \_\_mm\_mask\_unpackhi\_epi16(v s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKHWD \_\_m128i \_\_mm\_maskz\_unpackhi\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKHDQ \_\_m512i \_\_mm512\_unpackhi\_epi32( \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m512i \_\_mm512\_mask\_unpackhi\_epi32( \_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m512i \_\_mm512\_maskz\_unpackhi\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m256i \_\_mm256\_mask\_unpackhi\_epi32( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m256i \_\_mm256\_maskz\_unpackhi\_epi32( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m128i \_\_mm\_mask\_unpackhi\_epi32( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHDQ \_\_m128i \_\_mm\_maskz\_unpackhi\_epi32( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m512i \_\_mm512\_unpackhi\_epi64( \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m512i \_\_mm512\_mask\_unpackhi\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m512i \_\_mm512\_maskz\_unpackhi\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m256i \_\_mm256\_mask\_unpackhi\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m256i \_\_mm256\_maskz\_unpackhi\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m128i \_\_mm\_mask\_unpackhi\_epi64( \_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKHQDQ \_\_m128i \_\_mm\_maskz\_unpackhi\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 PUNPCKHBW: \_\_m64 \_\_mm\_unpackhi\_pi8( \_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKHBW: \_\_m128i \_\_mm\_unpackhi\_epi8( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKHBW: \_\_m256i \_\_mm256\_unpackhi\_epi8( \_\_m256i m1, \_\_m256i m2)  
 PUNPCKHWD: \_\_m64 \_\_mm\_unpackhi\_pi16( \_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKHWD: \_\_m128i \_\_mm\_unpackhi\_epi16( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKHWD: \_\_m256i \_\_mm256\_unpackhi\_epi16( \_\_m256i m1, \_\_m256i m2)  
 PUNPCKHDQ: \_\_m64 \_\_mm\_unpackhi\_pi32( \_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKHDQ: \_\_m128i \_\_mm\_unpackhi\_epi32( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKHDQ: \_\_m256i \_\_mm256\_unpackhi\_epi32( \_\_m256i m1, \_\_m256i m2)  
 (V)PUNPCKHQDQ: \_\_m128i \_\_mm\_unpackhi\_epi64 ( \_\_m128i a, \_\_m128i b)  
 VPUNPCKHQDQ: \_\_m256i \_\_mm256\_unpackhi\_epi64 ( \_\_m256i a, \_\_m256i b)

### Flags Affected

None.

### Numeric Exceptions

None.

### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPUNPCKHQDQ/QDQ, see Exceptions Type E4NF.

EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb.

## PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 60 /r <sup>1</sup> PUNPCKLBW <i>mm</i> , <i>mm/m32</i>	A	V/V	MMX	Interleave low-order bytes from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 OF 60 /r PUNPCKLBW <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Interleave low-order bytes from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
NP OF 61 /r <sup>1</sup> PUNPCKLWD <i>mm</i> , <i>mm/m32</i>	A	V/V	MMX	Interleave low-order words from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 OF 61 /r PUNPCKLWD <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Interleave low-order words from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
NP OF 62 /r <sup>1</sup> PUNPCKLDQ <i>mm</i> , <i>mm/m32</i>	A	V/V	MMX	Interleave low-order doublewords from <i>mm</i> and <i>mm/m32</i> into <i>mm</i> .
66 OF 62 /r PUNPCKLDQ <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Interleave low-order doublewords from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> .
66 OF 6C /r PUNPCKLQDQ <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Interleave low-order quadword from <i>xmm1</i> and <i>xmm2/m128</i> into <i>xmm1</i> register.
VEX.NDS.128.66.OF.WIG 60/r VPUNPCKLBW <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Interleave low-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 61/r VPUNPCKLWD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Interleave low-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 62/r VPUNPCKLDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Interleave low-order doublewords from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> .
VEX.NDS.128.66.OF.WIG 6C/r VPUNPCKLQDQ <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Interleave low-order quadword from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register.
VEX.NDS.256.66.OF.WIG 60 /r VPUNPCKLBW <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Interleave low-order bytes from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 61 /r VPUNPCKLWD <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Interleave low-order words from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 62 /r VPUNPCKLDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Interleave low-order doublewords from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
VEX.NDS.256.66.OF.WIG 6C /r VPUNPCKLQDQ <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Interleave low-order quadword from <i>ymm2</i> and <i>ymm3/m256</i> into <i>ymm1</i> register.
EVEX.NDS.128.66.OF.WIG 60 /r VPUNPCKLBW <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Interleave low-order bytes from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register subject to write mask <i>k1</i> .
EVEX.NDS.128.66.OF.WIG 61 /r VPUNPCKLWD <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128</i>	C	V/V	AVX512VL AVX512BW	Interleave low-order words from <i>xmm2</i> and <i>xmm3/m128</i> into <i>xmm1</i> register subject to write mask <i>k1</i> .
EVEX.NDS.128.66.OF.WO 62 /r VPUNPCKLDQ <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	D	V/V	AVX512VL AVX512F	Interleave low-order doublewords from <i>xmm2</i> and <i>xmm3/m128/m32bcst</i> into <i>xmm1</i> register subject to write mask <i>k1</i> .
EVEX.NDS.128.66.OF.W1 6C /r VPUNPCKLQDQ <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128/m64bcst</i>	D	V/V	AVX512VL AVX512F	Interleave low-order quadword from <i>zmm2</i> and <i>zmm3/m512/m64bcst</i> into <i>zmm1</i> register subject to write mask <i>k1</i> .

EVEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.W0 62 /r VPUNPCKLDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.W1 6C /r VPUNPCKLQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.WIG 60/r VPUNPCKLBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave low-order bytes from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.WIG 61/r VPUNPCKLWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave low-order words from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.W0 62 /r VPUNPCKLDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.W1 6C /r VPUNPCKLQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

**NOTES:**

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-22 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

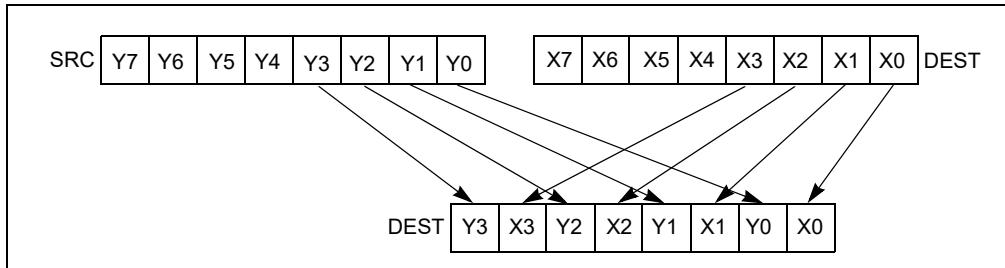


Figure 4-22. PUNPCKLBW Instruction Operation Using 64-bit Operands

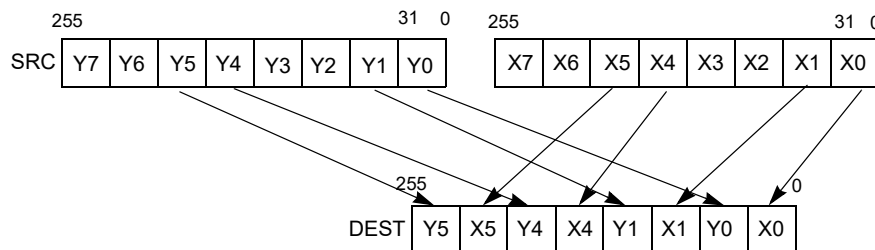


Figure 4-23. 256-bit VPUNPCKLDQ Instruction Operation

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

**Legacy SSE versions 64-bit operand:** The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

**128-bit Legacy SSE versions:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

**VEX.128 encoded versions:** The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

**VEX.256 encoded version:** The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

## Operation

### PUNPCKLBW instruction with 64-bit operands:

```
DEST[63:56] ← SRC[31:24];
DEST[55:48] ← DEST[31:24];
DEST[47:40] ← SRC[23:16];
DEST[39:32] ← DEST[23:16];
DEST[31:24] ← SRC[15:8];
DEST[23:16] ← DEST[15:8];
DEST[15:8] ← SRC[7:0];
DEST[7:0] ← DEST[7:0];
```

### PUNPCKLWD instruction with 64-bit operands:

```
DEST[63:48] ← SRC[31:16];
DEST[47:32] ← DEST[31:16];
DEST[31:16] ← SRC[15:0];
DEST[15:0] ← DEST[15:0];
```

### PUNPCKLDQ instruction with 64-bit operands:

```
DEST[63:32] ← SRC[31:0];
DEST[31:0] ← DEST[31:0];
```

INTERLEAVE\_BYTES\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_BYTES\_256b(SRC1[255:0], SRC[255:0])

TMP\_DEST[511:256] ← INTERLEAVE\_BYTES\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_BYTES\_256b (SRC1, SRC2)

```
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC1[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]
DEST[135:128] ← SRC1[135:128]
DEST[143:136] ← SRC2[135:128]
DEST[151:144] ← SRC1[143:136]
DEST[159:152] ← SRC2[143:136]
DEST[167:160] ← SRC1[151:144]
```

DEST[175:168] ← SRC2[151:144]  
 DEST[183:176] ← SRC1[159:152]  
 DEST[191:184] ← SRC2[159:152]  
 DEST[199:192] ← SRC1[167:160]  
 DEST[207:200] ← SRC2[167:160]  
 DEST[215:208] ← SRC1[175:168]  
 DEST[223:216] ← SRC2[175:168]  
 DEST[231:224] ← SRC1[183:176]  
 DEST[239:232] ← SRC2[183:176]  
 DEST[247:240] ← SRC1[191:184]  
 DEST[255:248] ← SRC2[191:184]

INTERLEAVE\_BYTES (SRC1, SRC2)

DEST[7:0] ← SRC1[7:0]  
 DEST[15:8] ← SRC2[7:0]  
 DEST[23:16] ← SRC2[15:8]  
 DEST[31:24] ← SRC2[15:8]  
 DEST[39:32] ← SRC1[23:16]  
 DEST[47:40] ← SRC2[23:16]  
 DEST[55:48] ← SRC1[31:24]  
 DEST[63:56] ← SRC2[31:24]  
 DEST[71:64] ← SRC1[39:32]  
 DEST[79:72] ← SRC2[39:32]  
 DEST[87:80] ← SRC1[47:40]  
 DEST[95:88] ← SRC2[47:40]  
 DEST[103:96] ← SRC1[55:48]  
 DEST[111:104] ← SRC2[55:48]  
 DEST[119:112] ← SRC1[63:56]  
 DEST[127:120] ← SRC2[63:56]

INTERLEAVE\_WORDS\_512b (SRC1, SRC2)

TMP\_DEST[255:0] ← INTERLEAVE\_WORDS\_256b(SRC1[255:0], SRC[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_WORDS\_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE\_WORDS\_256b(SRC1, SRC2)

DEST[15:0] ← SRC1[15:0]  
 DEST[31:16] ← SRC2[15:0]  
 DEST[47:32] ← SRC1[31:16]  
 DEST[63:48] ← SRC2[31:16]  
 DEST[79:64] ← SRC1[47:32]  
 DEST[95:80] ← SRC2[47:32]  
 DEST[111:96] ← SRC1[63:48]  
 DEST[127:112] ← SRC2[63:48]  
 DEST[143:128] ← SRC1[143:128]  
 DEST[159:144] ← SRC2[143:128]  
 DEST[175:160] ← SRC1[159:144]  
 DEST[191:176] ← SRC2[159:144]  
 DEST[207:192] ← SRC1[175:160]  
 DEST[223:208] ← SRC2[175:160]  
 DEST[239:224] ← SRC1[191:176]  
 DEST[255:240] ← SRC2[191:176]

INTERLEAVE\_WORDS (SRC1, SRC2)

DEST[15:0] ← SRC1[15:0]

DEST[31:16] ← SRC2[15:0]  
 DEST[47:32] ← SRC1[31:16]  
 DEST[63:48] ← SRC2[31:16]  
 DEST[79:64] ← SRC1[47:32]  
 DEST[95:80] ← SRC2[47:32]  
 DEST[111:96] ← SRC1[63:48]  
 DEST[127:112] ← SRC2[63:48]

INTERLEAVE\_DWORDS\_512b (SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_DWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_DWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_DWORDS\_256b(SRC1, SRC2)  
 DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[159:128] ← SRC1[159:128]  
 DEST[191:160] ← SRC2[159:128]  
 DEST[223:192] ← SRC1[191:160]  
 DEST[255:224] ← SRC2[191:160]

INTERLEAVE\_DWORDS(SRC1, SRC2)  
 DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 INTERLEAVE\_QWORDS\_512b (SRC1, SRC2)  
 TMP\_DEST[255:0] ← INTERLEAVE\_QWORDS\_256b(SRC1[255:0], SRC2[255:0])  
 TMP\_DEST[511:256] ← INTERLEAVE\_QWORDS\_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE\_QWORDS\_256b(SRC1, SRC2)  
 DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]  
 DEST[191:128] ← SRC1[191:128]  
 DEST[255:192] ← SRC2[191:128]

INTERLEAVE\_QWORDS(SRC1, SRC2)  
 DEST[63:0] ← SRC1[63:0]  
 DEST[127:64] ← SRC2[63:0]

**PUNPCKLBW**

DEST[127:0] ← INTERLEAVE\_BYTES(DEST, SRC)  
 DEST[255:127] (Unmodified)

**VPUNPCKLBW (VEX.128 encoded instruction)**

DEST[127:0] ← INTERLEAVE\_BYTES(SRC1, SRC2)  
 DEST[MAXVL-1:127] ← 0

**VPUNPCKLBW (VEX.256 encoded instruction)**

DEST[255:0] ← INTERLEAVE\_BYTES\_256b(SRC1, SRC2)  
 DEST[MAXVL-1:256] ← 0

**VPUNPCKLBW (EVEX.512 encoded instruction)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_BYTES\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_BYTES\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TMP\_DEST[i+7:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

DEST[511:0] ← INTERLEAVE\_BYTES\_512b(SRC1, SRC2)

**PUNPCKLWD**

DEST[127:0] ← INTERLEAVE\_WORDS(DEST, SRC)

DEST[255:127] (Unmodified)

**VPUNPCKLWD (VEX.128 encoded instruction)**

DEST[127:0] ← INTERLEAVE\_WORDS(SRC1, SRC2)

DEST[MAXVL-1:127] ← 0

**VPUNPCKLWD (VEX.256 encoded instruction)**

DEST[255:0] ← INTERLEAVE\_WORDS\_256b(SRC1, SRC2)

DEST[MAXVL-1:256] ← 0

**VPUNPCKLWD (EVEX.512 encoded instruction)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

TMP\_DEST[VL-1:0] ← INTERLEAVE\_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 256

TMP\_DEST[VL-1:0] ← INTERLEAVE\_WORDS\_256b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

IF VL = 512

TMP\_DEST[VL-1:0] ← INTERLEAVE\_WORDS\_512b(SRC1[VL-1:0], SRC2[VL-1:0])

FI;

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*



```

    THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
DEST[511:0] ← INTERLEAVE_WORDS_512b(SRC1, SRC2)

```

**PUNPCKLDQ**

```

DEST[127:0] ← INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

```

**VPUNPCKLDQ (VEX.128 encoded instruction)**

```

DEST[127:0] ← INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

```

**VPUNPCKLDQ (VEX.256 encoded instruction)**

```

DEST[255:0] ← INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

```

**VPUNPCKLDQ (EVEX encoded instructions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 128
  TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
  TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
  TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

```

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking*         ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  FI;
ENDFOR;

```

```

ENDFOR
DEST[511:0] ← INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAXVL-1:VL] ← 0

```

#### PUNPCKLQDQ

```

DEST[127:0] ← INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

```

#### VPUNPCKLQDQ (VEX.128 encoded instruction)

```

DEST[127:0] ← INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

```

#### VPUNPCKLQDQ (VEX.256 encoded instruction)

```

DEST[255:0] ← INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0

```

#### VPUNPCKLQDQ (EVEX encoded instructions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalents

```

VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8(__mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m256i _mm256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);

```

VPUNPCKLBW \_\_m256i \_\_mm256\_maskz\_unpacklo\_epi8( \_\_mmask32 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLBW \_\_m128i \_\_mm\_mask\_unpacklo\_epi8(v s, \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLBW \_\_m128i \_\_mm\_maskz\_unpacklo\_epi8( \_\_mmask16 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLWD \_\_m512i \_\_mm512\_unpacklo\_epi16(\_\_m512i a, \_\_m512i b);  
 VPUNPCKLWD \_\_m512i \_\_mm512\_mask\_unpacklo\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLWD \_\_m512i \_\_mm512\_maskz\_unpacklo\_epi16( \_\_mmask32 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLWD \_\_m256i \_\_mm256\_mask\_unpacklo\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLWD \_\_m256i \_\_mm256\_maskz\_unpacklo\_epi16( \_\_mmask16 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLWD \_\_m128i \_\_mm\_mask\_unpacklo\_epi16(v s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLWD \_\_m128i \_\_mm\_maskz\_unpacklo\_epi16( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLDQ \_\_m512i \_\_mm512\_unpacklo\_epi32(\_\_m512i a, \_\_m512i b);  
 VPUNPCKLDQ \_\_m512i \_\_mm512\_mask\_unpacklo\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLDQ \_\_m512i \_\_mm512\_maskz\_unpacklo\_epi32( \_\_mmask16 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLDQ \_\_m256i \_\_mm256\_mask\_unpacklo\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLDQ \_\_m256i \_\_mm256\_maskz\_unpacklo\_epi32( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLDQ \_\_m128i \_\_mm\_mask\_unpacklo\_epi32(v s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLDQ \_\_m128i \_\_mm\_maskz\_unpacklo\_epi32( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLQDQ \_\_m512i \_\_mm512\_unpacklo\_epi64(\_\_m512i a, \_\_m512i b);  
 VPUNPCKLQDQ \_\_m512i \_\_mm512\_mask\_unpacklo\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLQDQ \_\_m512i \_\_mm512\_maskz\_unpacklo\_epi64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b);  
 VPUNPCKLQDQ \_\_m256i \_\_mm256\_mask\_unpacklo\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLQDQ \_\_m256i \_\_mm256\_maskz\_unpacklo\_epi64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b);  
 VPUNPCKLQDQ \_\_m128i \_\_mm\_mask\_unpacklo\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 VPUNPCKLQDQ \_\_m128i \_\_mm\_maskz\_unpacklo\_epi64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);  
 PUNPCKLBW: \_\_m64 \_\_mm\_unpacklo\_pi8( \_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKLBW: \_\_m128i \_\_mm\_unpacklo\_epi8( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKLBW: \_\_m256i \_\_mm256\_unpacklo\_epi8( \_\_m256i m1, \_\_m256i m2)  
 PUNPCKLWD: \_\_m64 \_\_mm\_unpacklo\_pi16( \_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKLWD: \_\_m128i \_\_mm\_unpacklo\_epi16( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKLWD: \_\_m256i \_\_mm256\_unpacklo\_epi16( \_\_m256i m1, \_\_m256i m2)  
 PUNPCKLDQ: \_\_m64 \_\_mm\_unpacklo\_pi32( \_\_m64 m1, \_\_m64 m2)  
 (V)PUNPCKLDQ: \_\_m128i \_\_mm\_unpacklo\_epi32( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKLDQ: \_\_m256i \_\_mm256\_unpacklo\_epi32( \_\_m256i m1, \_\_m256i m2)  
 (V)PUNPCKLDQ: \_\_m128i \_\_mm\_unpacklo\_epi64( \_\_m128i m1, \_\_m128i m2)  
 VPUNPCKLQDQ: \_\_m256i \_\_mm256\_unpacklo\_epi64( \_\_m256i m1, \_\_m256i m2)

## Flags Affected

None.

## Numeric Exceptions

None.

## Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPUNPCKLDQ/QDQ, see Exceptions Type E4NF.

EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb.

## PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FF /6	PUSH <i>r/m16</i>	M	Valid	Valid	Push <i>r/m16</i> .
FF /6	PUSH <i>r/m32</i>	M	N.E.	Valid	Push <i>r/m32</i> .
FF /6	PUSH <i>r/m64</i>	M	Valid	N.E.	Push <i>r/m64</i> .
50+ <i>rw</i>	PUSH <i>r16</i>	0	Valid	Valid	Push <i>r16</i> .
50+ <i>rd</i>	PUSH <i>r32</i>	0	N.E.	Valid	Push <i>r32</i> .
50+ <i>rd</i>	PUSH <i>r64</i>	0	Valid	N.E.	Push <i>r64</i> .
6A <i>ib</i>	PUSH <i>imm8</i>	I	Valid	Valid	Push <i>imm8</i> .
68 <i>iw</i>	PUSH <i>imm16</i>	I	Valid	Valid	Push <i>imm16</i> .
68 <i>id</i>	PUSH <i>imm32</i>	I	Valid	Valid	Push <i>imm32</i> .
0E	PUSH CS	Z0	Invalid	Valid	Push CS.
16	PUSH SS	Z0	Invalid	Valid	Push SS.
1E	PUSH DS	Z0	Invalid	Valid	Push DS.
06	PUSH ES	Z0	Invalid	Valid	Push ES.
0F A0	PUSH FS	Z0	Valid	Valid	Push FS.
0F A8	PUSH GS	Z0	Valid	Valid	Push GS.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA
0	opcode + rd (r)	NA	NA	NA
I	imm8/16/32	NA	NA	NA
Z0	NA	NA	NA	NA

### Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).  
The address size is used only when referencing a source operand in memory.
- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Core and Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

### IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

### Operation

(\* See Description section for possible sign-extension or zero-extension of source operand and for \*)

(\* a case in which the size of the memory store may be smaller than the instruction's operand size \*)

IF StackAddrSize = 64

THEN

IF OperandSize = 64

THEN

RSP ← RSP - 8;

Memory[SS:RSP] ← SRC; (\* push quadword \*)

ELSE IF OperandSize = 32

THEN

RSP ← RSP - 4;

Memory[SS:RSP] ← SRC; (\* push dword \*)

ELSE (\* OperandSize = 16 \*)

RSP ← RSP - 2;

Memory[SS:RSP] ← SRC; (\* push word \*)

FI;

ELSE IF StackAddrSize = 32

THEN

IF OperandSize = 64

THEN

ESP ← ESP - 8;

Memory[SS:ESP] ← SRC; (\* push quadword \*)

ELSE IF OperandSize = 32

THEN

ESP ← ESP - 4;

Memory[SS:ESP] ← SRC; (\* push dword \*)

ELSE (\* OperandSize = 16 \*)

ESP ← ESP - 2;

Memory[SS:ESP] ← SRC; (\* push word \*)

FI;

ELSE (\* StackAddrSize = 16 \*)

```

    IF OperandSize = 32
    THEN
        SP ← SP - 4;
        Memory[SS:SP] ← SRC;          (* push dword *)
    ELSE (* OperandSize = 16 *)
        SP ← SP - 2;
        Memory[SS:SP] ← SRC;          (* push word *)
    FI;
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit. If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used. If the PUSH is of CS, SS, DS, or ES.

**PUSHA/PUSHAD—Push All General-Purpose Registers**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
60	PUSHA	Z0	Invalid	Valid	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	Z0	Invalid	Valid	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the “Operation” section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

**Operation**

IF 64-bit Mode

THEN #UD

FI;

IF OperandSize = 32 (\* PUSHAD instruction \*)

THEN

Temp ← (ESP);  
 Push(EAX);  
 Push(ECX);  
 Push(EDX);  
 Push(EBX);  
 Push(Temp);  
 Push(EBP);  
 Push(ESI);  
 Push(EDI);

ELSE (\* OperandSize = 16, PUSHA instruction \*)

Temp ← (SP);  
 Push(AX);  
 Push(CX);  
 Push(DX);

Push(BX);  
 Push(Temp);  
 Push(BP);  
 Push(SI);  
 Push(DI);

FI;

### Flags Affected

None.

### Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#UD	If in 64-bit mode.
-----	--------------------



**PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register onto the Stack**

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9C	PUSHF	Z0	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	Z0	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	Z0	Valid	N.E.	Push RFLAGS.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFQ instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS register.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In 64-bit mode, the instruction's default operation is to decrement the stack pointer (RSP) by 8 and pushes RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the PUSHF/PUSHFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), PUSHF (but not PUSHFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. PUSHF, however, is not affected by CR4.PVI.)

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

**Operation**

IF (PE = 0) or (PE = 1 and ((VM = 0) or (VM = 1 and IOPL = 3)))

(\* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 \*)

THEN

IF OperandSize = 32

THEN

push (EFLAGS AND 00FCFFFFH);

(\* VM and RF bits are cleared in image stored on the stack \*)

ELSE

push (EFLAGS); (\* Lower 16 bits only \*)

```

FI;

ELSE IF 64-bit MODE (* In 64-bit Mode *)
  IF OperandSize = 64
    THEN
      push (RFLAGS AND 00000000_00FCFFFFH);
      (* VM and RF bits are cleared in image stored on the stack; *)
    ELSE
      push (EFLAGS); (* Lower 16 bits only *)
  FI;

ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
  IF (CR4.VME = 0) OR (OperandSize = 32)
    THEN #GP(0); (* Trap to virtual-8086 monitor *)
  ELSE
    tempFLAGS = EFLAGS[15:0];
    tempFLAGS[9] = tempFLAGS[19]; (* VIF replaces IF *)
    tempFlags[13:12] = 3; (* IOPL is set to 3 in image stored on the stack *)
    push (tempFLAGS);
  FI;
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
-----	-----------------------------

### Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

## PXOR—Logical Exclusive OR

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EF /r <sup>1</sup> PXOR <i>mm</i> , <i>mm/m64</i>	A	V/V	MMX	Bitwise XOR of <i>mm/m64</i> and <i>mm</i> .
66 0F EF /r PXOR <i>xmm1</i> , <i>xmm2/m128</i>	A	V/V	SSE2	Bitwise XOR of <i>xmm2/m128</i> and <i>xmm1</i> .
VEX.NDS.128.66.0F.WIG EF /r VPXOR <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m128</i>	B	V/V	AVX	Bitwise XOR of <i>xmm3/m128</i> and <i>xmm2</i> .
VEX.NDS.256.66.0F.WIG EF /r VPXOR <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i>	B	V/V	AVX2	Bitwise XOR of <i>ymm3/m256</i> and <i>ymm2</i> .
EVEX.NDS.128.66.0F.W0 EF /r VPXORD <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128/m32bcst</i>	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in <i>xmm2</i> and <i>xmm3/m128</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W0 EF /r VPXORD <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256/m32bcst</i>	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in <i>ymm2</i> and <i>ymm3/m256</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W0 EF /r VPXORD <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>zmm3/m512/m32bcst</i>	C	V/V	AVX512F	Bitwise XOR of packed doubleword integers in <i>zmm2</i> and <i>zmm3/m512/m32bcst</i> using writemask <i>k1</i> .
EVEX.NDS.128.66.0F.W1 EF /r VPXORQ <i>xmm1</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i> , <i>xmm3/m128/m64bcst</i>	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in <i>xmm2</i> and <i>xmm3/m128</i> using writemask <i>k1</i> .
EVEX.NDS.256.66.0F.W1 EF /r VPXORQ <i>ymm1</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i> , <i>ymm3/m256/m64bcst</i>	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in <i>ymm2</i> and <i>ymm3/m256</i> using writemask <i>k1</i> .
EVEX.NDS.512.66.0F.W1 EF /r VPXORQ <i>zmm1</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i> , <i>zmm3/m512/m64bcst</i>	C	V/V	AVX512F	Bitwise XOR of packed quadword integers in <i>zmm2</i> and <i>zmm3/m512/m64bcst</i> using writemask <i>k1</i> .

### NOTES:

1. See note in Section 2.4, “AVX and SSE Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 22.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM:r/m ( <i>r</i> )	NA	NA
B	NA	ModRM:reg ( <i>w</i> )	VEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA
C	Full	ModRM:reg ( <i>w</i> )	EVEX.vvvv ( <i>r</i> )	ModRM:r/m ( <i>r</i> )	NA

### Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

## Operation

### PXOR (64-bit operand)

DEST  $\leftarrow$  DEST XOR SRC

### PXOR (128-bit Legacy SSE version)

DEST  $\leftarrow$  DEST XOR SRC

DEST[MAXVL-1:128] (Unmodified)

### VPXOR (VEX.128 encoded version)

DEST  $\leftarrow$  SRC1 XOR SRC2

DEST[MAXVL-1:128]  $\leftarrow$  0

### VPXOR (VEX.256 encoded version)

DEST  $\leftarrow$  SRC1 XOR SRC2

DEST[MAXVL-1:256]  $\leftarrow$  0

### VPXORD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

      THEN DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE XOR SRC2[31:0]

      ELSE DEST[i+31:i]  $\leftarrow$  SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[31:0] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[31:0]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

DEST[MAXVL-1:VL]  $\leftarrow$  0

**VPXORQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0]

ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPXORD \_\_m512i \_mm512\_xor\_epi32(\_\_m512i a, \_\_m512i b)

VPXORD \_\_m512i \_mm512\_mask\_xor\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a, \_\_m512i b)

VPXORD \_\_m512i \_mm512\_maskz\_xor\_epi32(\_\_mmask16 m, \_\_m512i a, \_\_m512i b)

VPXORD \_\_m256i \_mm256\_xor\_epi32(\_\_m256i a, \_\_m256i b)

VPXORD \_\_m256i \_mm256\_mask\_xor\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i b)

VPXORD \_\_m256i \_mm256\_maskz\_xor\_epi32(\_\_mmask8 m, \_\_m256i a, \_\_m256i b)

VPXORD \_\_m128i \_mm\_xor\_epi32(\_\_m128i a, \_\_m128i b)

VPXORD \_\_m128i \_mm\_mask\_xor\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b)

VPXORD \_\_m128i \_mm\_maskz\_xor\_epi32(\_\_mmask16 m, \_\_m128i a, \_\_m128i b)

VPXORQ \_\_m512i \_mm512\_xor\_epi64(\_\_m512i a, \_\_m512i b);

VPXORQ \_\_m512i \_mm512\_mask\_xor\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a, \_\_m512i b);

VPXORQ \_\_m512i \_mm512\_maskz\_xor\_epi64(\_\_mmask8 m, \_\_m512i a, \_\_m512i b);

VPXORQ \_\_m256i \_mm256\_xor\_epi64(\_\_m256i a, \_\_m256i b);

VPXORQ \_\_m256i \_mm256\_mask\_xor\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPXORQ \_\_m256i \_mm256\_maskz\_xor\_epi64(\_\_mmask8 m, \_\_m256i a, \_\_m256i b);

VPXORQ \_\_m128i \_mm\_xor\_epi64(\_\_m128i a, \_\_m128i b);

VPXORQ \_\_m128i \_mm\_mask\_xor\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b);

VPXORQ \_\_m128i \_mm\_maskz\_xor\_epi64(\_\_mmask8 m, \_\_m128i a, \_\_m128i b);

PXOR: \_\_m64 \_mm\_xor\_si64 (\_\_m64 m1, \_\_m64 m2)

(V)PXOR: \_\_m128i \_mm\_xor\_si128 (\_\_m128i a, \_\_m128i b)

VPXOR: \_\_m256i \_mm256\_xor\_si256 (\_\_m256i a, \_\_m256i b)

**Flags Affected**

None.

**Numeric Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

**RCL/RCR/ROL/ROR—Rotate**

Opcode**	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
D0 /2	RCL <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) left once.
REX + D0 /2	RCL <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) left once.
D2 /2	RCL <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) left CL times.
REX + D2 /2	RCL <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) left CL times.
C0 /2 <i>ib</i>	RCL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) left <i>imm8</i> times.
REX + C0 /2 <i>ib</i>	RCL <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) left <i>imm8</i> times.
D1 /2	RCL <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) left once.
D3 /2	RCL <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) left CL times.
C1 /2 <i>ib</i>	RCL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) left <i>imm8</i> times.
D1 /2	RCL <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) left once.
REX.W + D1 /2	RCL <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) left once. Uses a 6 bit count.
D3 /2	RCL <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) left CL times.
REX.W + D3 /2	RCL <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) left CL times. Uses a 6 bit count.
C1 /2 <i>ib</i>	RCL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) left <i>imm8</i> times.
REX.W + C1 /2 <i>ib</i>	RCL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) left <i>imm8</i> times. Uses a 6 bit count.
D0 /3	RCR <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) right once.
REX + D0 /3	RCR <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) right once.
D2 /3	RCR <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) right CL times.
REX + D2 /3	RCR <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) right CL times.
C0 /3 <i>ib</i>	RCR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 9 bits (CF, <i>r/m8</i> ) right <i>imm8</i> times.
REX + C0 /3 <i>ib</i>	RCR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 9 bits (CF, <i>r/m8</i> ) right <i>imm8</i> times.
D1 /3	RCR <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) right once.
D3 /3	RCR <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) right CL times.
C1 /3 <i>ib</i>	RCR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 17 bits (CF, <i>r/m16</i> ) right <i>imm8</i> times.
D1 /3	RCR <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) right once. Uses a 6 bit count.
REX.W + D1 /3	RCR <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) right once. Uses a 6 bit count.
D3 /3	RCR <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) right CL times.
REX.W + D3 /3	RCR <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) right CL times. Uses a 6 bit count.
C1 /3 <i>ib</i>	RCR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 33 bits (CF, <i>r/m32</i> ) right <i>imm8</i> times.
REX.W + C1 /3 <i>ib</i>	RCR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 65 bits (CF, <i>r/m64</i> ) right <i>imm8</i> times. Uses a 6 bit count.
D0 /0	ROL <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 8 bits <i>r/m8</i> left once.
REX + D0 /0	ROL <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left once
D2 /0	ROL <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 8 bits <i>r/m8</i> left CL times.
REX + D2 /0	ROL <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left CL times.
C0 /0 <i>ib</i>	ROL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 8 bits <i>r/m8</i> left <i>imm8</i> times.

Opcode**	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
REX + C0 /0 <i>ib</i>	ROL <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 8 bits <i>r/m8</i> left <i>imm8</i> times.
D1 /0	ROL <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 16 bits <i>r/m16</i> left once.
D3 /0	ROL <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 16 bits <i>r/m16</i> left CL times.
C1 /0 <i>ib</i>	ROL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 16 bits <i>r/m16</i> left <i>imm8</i> times.
D1 /0	ROL <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 32 bits <i>r/m32</i> left once.
REX.W + D1 /0	ROL <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left once. Uses a 6 bit count.
D3 /0	ROL <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 32 bits <i>r/m32</i> left CL times.
REX.W + D3 /0	ROL <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left CL times. Uses a 6 bit count.
C1 /0 <i>ib</i>	ROL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 32 bits <i>r/m32</i> left <i>imm8</i> times.
REX.W + C1 /0 <i>ib</i>	ROL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 64 bits <i>r/m64</i> left <i>imm8</i> times. Uses a 6 bit count.
D0 /1	ROR <i>r/m8</i> , 1	M1	Valid	Valid	Rotate 8 bits <i>r/m8</i> right once.
REX + D0 /1	ROR <i>r/m8*</i> , 1	M1	Valid	N.E.	Rotate 8 bits <i>r/m8</i> right once.
D2 /1	ROR <i>r/m8</i> , CL	MC	Valid	Valid	Rotate 8 bits <i>r/m8</i> right CL times.
REX + D2 /1	ROR <i>r/m8*</i> , CL	MC	Valid	N.E.	Rotate 8 bits <i>r/m8</i> right CL times.
C0 /1 <i>ib</i>	ROR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 8 bits <i>r/m16</i> right <i>imm8</i> times.
REX + C0 /1 <i>ib</i>	ROR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 8 bits <i>r/m16</i> right <i>imm8</i> times.
D1 /1	ROR <i>r/m16</i> , 1	M1	Valid	Valid	Rotate 16 bits <i>r/m16</i> right once.
D3 /1	ROR <i>r/m16</i> , CL	MC	Valid	Valid	Rotate 16 bits <i>r/m16</i> right CL times.
C1 /1 <i>ib</i>	ROR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 16 bits <i>r/m16</i> right <i>imm8</i> times.
D1 /1	ROR <i>r/m32</i> , 1	M1	Valid	Valid	Rotate 32 bits <i>r/m32</i> right once.
REX.W + D1 /1	ROR <i>r/m64</i> , 1	M1	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right once. Uses a 6 bit count.
D3 /1	ROR <i>r/m32</i> , CL	MC	Valid	Valid	Rotate 32 bits <i>r/m32</i> right CL times.
REX.W + D3 /1	ROR <i>r/m64</i> , CL	MC	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right CL times. Uses a 6 bit count.
C1 /1 <i>ib</i>	ROR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Rotate 32 bits <i>r/m32</i> right <i>imm8</i> times.
REX.W + C1 /1 <i>ib</i>	ROR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Rotate 64 bits <i>r/m64</i> right <i>imm8</i> times. Uses a 6 bit count.

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

\*\* See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (w)	1	NA	NA
MC	ModRM:r/m (w)	CL	NA	NA
MI	ModRM:r/m (w)	imm8	NA	NA

## Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W = 1).

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except RCL and RCR instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

## IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

## Operation

(\* RCL and RCR instructions \*)

SIZE ← OperandSize;

CASE (determine count) OF

SIZE ← 8: tempCOUNT ← (COUNT AND 1FH) MOD 9;  
 SIZE ← 16: tempCOUNT ← (COUNT AND 1FH) MOD 17;  
 SIZE ← 32: tempCOUNT ← COUNT AND 1FH;  
 SIZE ← 64: tempCOUNT ← COUNT AND 3FH;

ESAC;

(\* RCL instruction operation \*)

WHILE (tempCOUNT ≠ 0)

DO

tempCF ← MSB(DEST);  
 DEST ← (DEST \* 2) + CF;  
 CF ← tempCF;  
 tempCOUNT ← tempCOUNT - 1;

OD;

ELIHW;

IF (COUNT & COUNTMASK) = 1

THEN OF ← MSB(DEST) XOR CF;  
 ELSE OF is undefined;

FI;



(\* RCR instruction operation \*)

IF (COUNT & COUNTMASK) = 1

THEN OF  $\leftarrow$  MSB(DEST) XOR CF;

ELSE OF is undefined;

FI;

WHILE (tempCOUNT  $\neq$  0)

DO

tempCF  $\leftarrow$  LSB(SRC);

DEST  $\leftarrow$  (DEST / 2) + (CF \*  $2^{\text{SIZE}}$ );

CF  $\leftarrow$  tempCF;

tempCOUNT  $\leftarrow$  tempCOUNT - 1;

OD;

(\* ROL and ROR instructions \*)

IF OperandSize = 64

THEN COUNTMASK = 3FH;

ELSE COUNTMASK = 1FH;

FI;

(\* ROL instruction operation \*)

tempCOUNT  $\leftarrow$  (COUNT & COUNTMASK) MOD SIZE

WHILE (tempCOUNT  $\neq$  0)

DO

tempCF  $\leftarrow$  MSB(DEST);

DEST  $\leftarrow$  (DEST \* 2) + tempCF;

tempCOUNT  $\leftarrow$  tempCOUNT - 1;

OD;

ELIHW;

IF (COUNT & COUNTMASK)  $\neq$  0

THEN CF  $\leftarrow$  LSB(DEST);

FI;

IF (COUNT & COUNTMASK) = 1

THEN OF  $\leftarrow$  MSB(DEST) XOR CF;

ELSE OF is undefined;

FI;

(\* ROR instruction operation \*)

tempCOUNT  $\leftarrow$  (COUNT & COUNTMASK) MOD SIZE

WHILE (tempCOUNT  $\neq$  0)

DO

tempCF  $\leftarrow$  LSB(SRC);

DEST  $\leftarrow$  (DEST / 2) + (tempCF \*  $2^{\text{SIZE}}$ );

tempCOUNT  $\leftarrow$  tempCOUNT - 1;

OD;

ELIHW;

IF (COUNT & COUNTMASK)  $\neq$  0

THEN CF  $\leftarrow$  MSB(DEST);

FI;

IF (COUNT & COUNTMASK) = 1

THEN OF  $\leftarrow$  MSB(DEST) XOR MSB - 1(DEST);

ELSE OF is undefined;

FI;

## Flags Affected

If the masked count is 0, the flags are not affected. If the masked count is 1, then the OF flag is affected, otherwise (masked count is greater than 1) the OF flag is undefined. The CF flag is affected when the masked count is non-zero. The SF, ZF, AF, and PF flags are always unaffected.

## Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the source operand is located in a nonwritable segment. If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 53 /r RCPPS <i>xmm1, xmm2/m128</i>	RM	V/V	SSE	Computes the approximate reciprocals of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.0F.WIG 53 /r VRCPPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in <i>xmm2/mem</i> and stores the results in <i>xmm1</i> .
VEX.256.0F.WIG 53 /r VRCPPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in <i>ymm2/mem</i> and stores the results in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to  $|1.1111111110100000000000B * 2^{125}|$  are guaranteed to not produce tiny results; input values less than or equal to  $|1.0000000000110000000001B * 2^{126}|$  are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

## Operation

### RCPPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[127:96])$   
 $\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

### VRCPPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[127:96])$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$

### VRCPPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[31:0])$   
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[63:32])$   
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[95:64])$   
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[127:96])$   
 $\text{DEST}[159:128] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[159:128])$   
 $\text{DEST}[191:160] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[191:160])$   
 $\text{DEST}[223:192] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[223:192])$   
 $\text{DEST}[255:224] \leftarrow \text{APPROXIMATE}(1/\text{SRC}[255:224])$

## Intel C/C++ Compiler Intrinsic Equivalent

RCCPS: `__m128 _mm_rcp_ps(__m128 a)`  
 RCPPS: `__m256 _mm256_rcp_ps (__m256 a);`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv  $\neq$  1111B.

## RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 53 /r RCPSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Computes the approximate reciprocal of the scalar single-precision floating-point value in <i>xmm2/m32</i> and stores the result in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 53 /r VRCPSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Computes the approximate reciprocal of the scalar single-precision floating-point value in <i>xmm3/m32</i> and stores the result in <i>xmm1</i> . Also, upper single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes an approximate reciprocal of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to  $|1.1111111110100000000000B * 2^{125}|$  are guaranteed to not produce tiny results; input values less than or equal to  $|1.0000000000110000000001B * 2^{126}|$  are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

### Operation

#### RCPSS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SRC[31:0])

DEST[MAXVL-1:32] (Unmodified)

**VRC PSS (VEX.128 encoded version)** $\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SRC2}[31:0])$  $\text{DEST}[127:32] \leftarrow \text{SRC1}[127:32]$  $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$ **Intel C/C++ Compiler Intrinsic Equivalent**RCPSS: `__m128 _mm_rcp_ss(__m128 a)`**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5.

**RDFSBASE/RDGSBASE—Read FS/GS Segment Base**

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Fea- ture Flag	Description
F3 0F AE /0 RDFSBASE r32	M	V/I	FSGSBASE	Load the 32-bit destination register with the FS base address.
F3 REX.W 0F AE /0 RDFSBASE r64	M	V/I	FSGSBASE	Load the 64-bit destination register with the FS base address.
F3 0F AE /1 RDGSBASE r32	M	V/I	FSGSBASE	Load the 32-bit destination register with the GS base address.
F3 REX.W 0F AE /1 RDGSBASE r64	M	V/I	FSGSBASE	Load the 64-bit destination register with the GS base address.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

**Description**

Loads the general-purpose register indicated by the modR/M:r/m field with the FS or GS segment base address.

The destination operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source base address (for FS or GS) are ignored and upper 32 bits of the destination register are cleared.

This instruction is supported only in 64-bit mode.

**Operation**

DEST ← FS/GS segment base address;

**Flags Affected**

None

**C/C++ Compiler Intrinsic Equivalent**

```

RDFSBASE:    unsigned int _readfsbase_u32(void);
RDFSBASE:    unsigned __int64 _readfsbase_u64(void);
RDGSBASE:    unsigned int _readgsbase_u32(void);
RDGSBASE:    unsigned __int64 _readgsbase_u64(void);

```

**Protected Mode Exceptions**

#UD The RDFSBASE and RDGSBASE instructions are not recognized in protected mode.

**Real-Address Mode Exceptions**

#UD The RDFSBASE and RDGSBASE instructions are not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD The RDFSBASE and RDGSBASE instructions are not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

#UD The RDFSBASE and RDGSBASE instructions are not recognized in compatibility mode.

## 64-Bit Mode Exceptions

#UD                    If the LOCK prefix is used.  
                      If CR4.FSGSBASE[bit 16] = 0.  
                      If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0.



## RDMSR—Read from Model Specific Register

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 32	RDMSR	Z0	Valid	Valid	Read MSR specified by ECX into EDX:EAX.

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Reads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Chapter 2, “Model-Specific Registers (MSRs)” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

EDX:EAX ← MSR[ECX];

### Flags Affected

None.

### Protected Mode Exceptions

- |        |  |
|--------|--|
| #GP(0) | If the current privilege level is not 0.                               |
|        | If the value in ECX specifies a reserved or unimplemented MSR address. |
| #UD    | If the LOCK prefix is used.  |

**Real-Address Mode Exceptions**

- #GP                      If the value in ECX specifies a reserved or unimplemented MSR address.  
#UD                      If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

- #GP(0)                  The RDMSR instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

**RDPID—Read Processor ID**

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F C7 /7 RDPID r32	R	N.E./V	RDPID	Read IA32_TSC_AUX into r32.
F3 0F C7 /7 RDPID r64	R	V/N.E.	RDPID	Read IA32_TSC_AUX into r64.

**Instruction Operand Encoding<sup>1</sup>**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
R	ModRM:r/m (w)	NA	NA	NA

**Description**

Reads the value of the IA32\_TSC\_AUX MSR (address C0000103H) into the destination register. The value of CS.D and operand-size prefixes (66H and REX.W) do not affect the behavior of the RDPID instruction.

**Operation**

DEST  $\leftarrow$  IA32\_TSC\_AUX

**Flags Affected**

None.

**Protected Mode Exceptions**

#UD If the LOCK prefix is used.  
If CPUID.7H.0:ECX.RDPID[bit 22] = 0.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

---

1. ModRM.MOD = 011B required

## RDPKRU—Read Protection Key Rights for User Pages

Opcode*	Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EE	RDPKRU	Z0	V/V	OSPKE	Reads PKRU into EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Reads the value of PKRU into EAX and clears EDX. ECX must be 0 when RDPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

RDPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX are ignored and the high-order 32-bits of RDX and RAX are cleared.

### Operation

```
IF (ECX = 0)
    THEN
        EAX ← PKRU;
        EDX ← 0;
    ELSE #GP(0);
FI;
```

### Flags Affected

None.

### C/C++ Compiler Intrinsic Equivalent

```
RDPKRU:      uint32_t _rdpkru_u32(void);
```

### Protected Mode Exceptions

#GP(0)                If ECX ≠ 0  
 #UD                 If the LOCK prefix is used.  
                       If CR4.PKE = 0.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## RDPMC—Read Performance-Monitoring Counters

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 33	RDPMC	Z0	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

The EAX register is loaded with the low-order 32 bits. The EDX register is loaded with the supported high-order bits of the counter. The number of high-order bits loaded into EDX is implementation specific on processors that do not support architectural performance monitoring. The width of fixed-function and general-purpose performance counters on processors supporting architectural performance monitoring are reported by CPUID 0AH leaf. See below for the treatment of the EDX register for “fast” reads.

The ECX register specifies the counter type (if the processor supports architectural performance monitoring) and counter index. Counter type is specified in ECX[30] to select one of two type of performance counters. If the processor does not support architectural performance monitoring, ECX[30:0] specifies the counter index; otherwise ECX[29:0] specifies the index relative to the base of each counter type. ECX[31] selects “fast” read mode if supported. The two counter types are:

- General-purpose or special-purpose performance counters are specified with ECX[30] = 0: The number of general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf. The number of general-purpose counters is model specific if the processor does not support architectural performance monitoring, see Chapter 18, “Performance Monitoring” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. Special-purpose counters are available only in selected processor members, see Table 4-16.
- Fixed-function performance counters are specified with ECX[30] = 1. The number fixed-function performance counters is enumerated by CPUID 0AH leaf. See Chapter 18, “Performance Monitoring” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*. This counter type is selected if ECX[30] is set.

The width of fixed-function performance counters and general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf. The width of general-purpose performance counters are 40-bits for processors that do not support architectural performance monitoring counters. The width of special-purpose performance counters are implementation specific.

Table 4-16 lists valid indices of the general-purpose and special-purpose performance counters according to the DisplayFamily\_DisplayModel values of CPUID encoding for each processor family (see CPUID instruction in Chapter 3, “Instruction Set Reference, A-L” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*).

**Table 4-16. Valid General and Special Purpose Performance Counter Index Range for RDPMC**

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
P6	06H_01H, 06H_03H, 06H_05H, 06H_06H, 06H_07H, 06H_08H, 06H_0AH, 06H_0BH	0, 1	0, 1
Processors Based on Intel NetBurst microarchitecture (No L3)	0FH_00H, 0FH_01H, 0FH_02H, 0FH_03H, 0FH_04H, 0FH_06H	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium M processors	06H_09H, 06H_0DH	0, 1	0, 1
Processors Based on Intel NetBurst microarchitecture (No L3)	0FH_03H, 0FH_04H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17

**Table 4-16. Valid General and Special Purpose Performance Counter Index Range for RDPMC (Contd.)**

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
Intel® Core™ Solo and Intel® Core™ Duo processors, Dual-core Intel® Xeon® processor LV	06H_0EH	0, 1	0, 1
Intel® Core™2 Duo processor, Intel Xeon processor 3000, 5100, 5300, 7300 Series - general-purpose PMC	06H_0FH	0, 1	0, 1
Intel® Core™2 Duo processor family, Intel Xeon processor 3100, 3300, 5200, 5400 series - general-purpose PMC	06H_17H	0, 1	0, 1
Intel Xeon processors 7400 series	(06H_1DH)	≥ 0 and ≤ 9	0, 1
45 nm and 32 nm Intel® Atom™ processors	06H_1CH, 06_26H, 06_27H, 06_35H, 06_36H	0, 1	0, 1
Intel® Atom™ processors based on Silvermont or Airmont microarchitectures	06H_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_4CH	0, 1	0, 1
Next Generation Intel® Atom™ processors based on Goldmont microarchitecture	06H_5CH, 06_5FH	0-3	0-3
Intel® processors based on the Nehalem, Westmere microarchitectures	06H_1AH, 06H_1EH, 06H_1FH, 06_25H, 06_2CH, 06H_2EH, 06_2FH	0-3	0-3
Intel® processors based on the Sandy Bridge, Ivy Bridge microarchitecture	06H_2AH, 06H_2DH, 06H_3AH, 06H_3EH	0-3 (0-7 if HyperThreading is off)	0-3 (0-7 if HyperThreading is off)
Intel® processors based on the Haswell, Broadwell, Skylake microarchitectures	06H_3CH, 06H_45H, 06H_46H, 06H_3FH, 06_3DH, 06_47H, 4FH, 06_56H, 06_4EH, 06_5EH	0-3 (0-7 if HyperThreading is off)	0-3 (0-7 if HyperThreading is off)

Processors based on Intel NetBurst microarchitecture support “fast” (32-bit) and “slow” (40-bit) reads on the first 18 performance counters. Selected this option using ECX[31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on these processors than a full 40-bit read.

On processors based on Intel NetBurst microarchitecture with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters.

In Intel Core 2 processor family, Intel Xeon processor 3000, 5100, 5300 and 7400 series, the fixed-function performance counters are 40-bits wide; they can be accessed by RDPMC with ECX between from 4000\_0000H and 4000\_0002H.

On Intel Xeon processor 7400 series, there are eight 32-bit special-purpose counters addressable with indices 2-9, ECX[30]=0.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Chapter 19, “Performance Monitoring Events,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B*, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If

an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

Performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

## Operation

(\* Intel processors that support architectural performance monitoring \*)

Most significant counter bit (MSCB) = 47

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
  THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
    EAX ← IA32_FIXED_CTR(ECX)[30:0];
    EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
  ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0];
    EDX ← PMC(ECX[30:0])[MSCB:32];
  ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
    #GP(0);
```

FI;

(\* Intel Core 2 Duo processor family and Intel Xeon processor 3000, 5100, 5300, 7400 series\*)

Most significant counter bit (MSCB) = 39

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
  THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
    EAX ← IA32_FIXED_CTR(ECX)[30:0];
    EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
  ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0];
    EDX ← PMC(ECX[30:0])[MSCB:32];
  ELSE IF (ECX[30] = 0 and ECX[29:0] in valid special-purpose counter range)
    EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
  ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
    #GP(0);
```

FI;

(\* P6 family processors and Pentium processor with MMX technology \*)

```
IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
  THEN
    EAX ← PMC(ECX)[31:0];
    EDX ← PMC(ECX)[39:32];
  ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
    #GP(0);
```

FI;

(\* Processors based on Intel NetBurst microarchitecture \*)

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
  THEN IF (ECX[30:0] = 0:17)
    THEN IF ECX[31] = 0
```



```

    THEN
        EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
        EDX ← PMC(ECX[30:0])[39:32];
    ELSE (* ECX[31] = 1 *)
        THEN
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE IF (* 64-bit Intel processor based on Intel NetBurst microarchitecture with L3 *)
        THEN IF (ECX[30:0] = 18:25 )
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE (* Invalid PMC index in ECX[30:0], see Table 4-19. *)
        GP(0);
    FI;
ELSE (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.  
 If an invalid performance counter index is specified (see Table 4-16).  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP If an invalid performance counter index is specified (see Table 4-16).  
 #UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0) If the PCE flag in the CR4 register is clear.  
 If an invalid performance counter index is specified (see Table 4-16).  
 #UD If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0) If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.  
 If an invalid performance counter index is specified (see Table 4-16).  
 #UD If the LOCK prefix is used.

## RDRAND—Read Random Number

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C7 /6 RDRAND r16	M	V/V	RDRAND	Read a 16-bit random number and store in the destination register.
OF C7 /6 RDRAND r32	M	V/V	RDRAND	Read a 32-bit random number and store in the destination register.
REX.W + OF C7 /6 RDRAND r64	M	V/I	RDRAND	Read a 64-bit random number and store in the destination register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Loads a hardware generated random value and store it in the destination register. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random value has been returned, otherwise it is expected to loop and retry execution of RDRAND (see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, Section 7.3.17, "Random Number Generator Instructions").

This instruction is available at all privilege levels.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```

IF HW_RND_GEN.ready = 1
  THEN
    CASE of
      osize is 64: DEST[63:0] ← HW_RND_GEN.data;
      osize is 32: DEST[31:0] ← HW_RND_GEN.data;
      osize is 16: DEST[15:0] ← HW_RND_GEN.data;
    ESAC
    CF ← 1;
  ELSE
    CASE of
      osize is 64: DEST[63:0] ← 0;
      osize is 32: DEST[31:0] ← 0;
      osize is 16: DEST[15:0] ← 0;
    ESAC
    CF ← 0;
  FI
OF, SF, ZF, AF, PF ← 0;

```

### Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

### Intel C/C++ Compiler Intrinsic Equivalent

RDRAND:        `int _rdrand16_step( unsigned short * );`  
RDRAND:        `int _rdrand32_step( unsigned int * );`  
RDRAND:        `int _rdrand64_step( unsigned __int64 * );`

### Protected Mode Exceptions

#UD                If the LOCK prefix is used.  
                    If the F2H or F3H prefix is used.  
                    If CPUID.01H:ECX.RDRAND[bit 30] = 0.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## RDSEED—Read Random SEED

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF C7 /7 RDSEED r16	M	V/V	RDSEED	Read a 16-bit NIST SP800-90B & C compliant random value and store in the destination register.
OF C7 /7 RDSEED r32	M	V/V	RDSEED	Read a 32-bit NIST SP800-90B & C compliant random value and store in the destination register.
REX.W + OF C7 /7 RDSEED r64	M	V/I	RDSEED	Read a 64-bit NIST SP800-90B & C compliant random value and store in the destination register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Loads a hardware generated random value and store it in the destination register. The random value is generated from an Enhanced NRBG (Non Deterministic Random Bit Generator) that is compliant to NIST SP800-90B and NIST SP800-90C in the XOR construction mode. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random seed value has been returned, otherwise it is expected to loop and retry execution of RDSEED (see Section 1.2).

The RDSEED instruction is available at all privilege levels. The RDSEED instruction executes normally either inside or outside a transaction region.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

IF HW\_NRND\_GEN.ready = 1

THEN

CASE of

osize is 64: DEST[63:0] ← HW\_NRND\_GEN.data;

osize is 32: DEST[31:0] ← HW\_NRND\_GEN.data;

osize is 16: DEST[15:0] ← HW\_NRND\_GEN.data;

ESAC;

CF ← 1;

ELSE

CASE of

osize is 64: DEST[63:0] ← 0;

osize is 32: DEST[31:0] ← 0;

osize is 16: DEST[15:0] ← 0;

ESAC;

CF ← 0;

FI;

OF, SF, ZF, AF, PF ← 0;

## Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

## C/C++ Compiler Intrinsic Equivalent

```
RDSEED int _rdseed16_step( unsigned short * );
```

```
RDSEED int _rdseed32_step( unsigned int * );
```

```
RDSEED int _rdseed64_step( unsigned __int64 * );
```

## Protected Mode Exceptions

#UD                    If the LOCK prefix is used.  
                       If the F2H or F3H prefix is used.  
                       If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

## Real-Address Mode Exceptions

#UD                    If the LOCK prefix is used.  
                       If the F2H or F3H prefix is used.  
                       If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

## Virtual-8086 Mode Exceptions

#UD                    If the LOCK prefix is used.  
                       If the F2H or F3H prefix is used.  
                       If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

## Compatibility Mode Exceptions

#UD                    If the LOCK prefix is used.  
                       If the F2H or F3H prefix is used.  
                       If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

## 64-Bit Mode Exceptions

#UD                    If the LOCK prefix is used.  
                       If the F2H or F3H prefix is used.  
                       If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

## RDTSC—Read Time-Stamp Counter

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 31	RDTSC	ZO	Valid	Valid	Read time-stamp counter into EDX:EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

### Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. If software requires RDTSC to be executed only after all previous instructions have completed locally, it can either use RDTSCP (if the processor supports that instruction) or execute the sequence LFENCE;RDTSC.

This instruction was introduced by the Pentium processor.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN EDX:EAX ← TimeStampCounter;
ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## RDTSCP—Read Time-Stamp Counter and Processor ID

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 F9	RDTSCP	Z0	Valid	Valid	Read 64-bit time-stamp counter and IA32_TSC_AUX value into EDX:EAX and ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also reads the value of the IA32\_TSC\_AUX MSR (address C0000103H) into the ECX register. The EDX register is loaded with the high-order 32 bits of the IA32\_TSC MSR; the EAX register is loaded with the low-order 32 bits of the IA32\_TSC MSR; and the ECX register is loaded with the low-order 32-bits of IA32\_TSC\_AUX MSR. On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX, RDX, and RCX are cleared.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSCP instruction as follows. When the flag is clear, the RDTSCP instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The RDTSCP instruction waits until all previous instructions have been executed before reading the counter. However, subsequent instructions may begin execution before the read operation is performed.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN
        EDX:EAX ← TimeStampCounter;
        ECX ← IA32_TSC_AUX[31:0];
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the TSD flag in register CR4 is set and the CPL is greater than 0.  
 #UD If the LOCK prefix is used.  
 If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

### Real-Address Mode Exceptions

#UD If the LOCK prefix is used.  
 If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.



### Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used. If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 6C	REP INS <i>m8, DX</i>	Z0	Valid	Valid	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6C	REP INS <i>m8, DX</i>	Z0	Valid	N.E.	Input RCX bytes from port DX into [RDI].
F3 6D	REP INS <i>m16, DX</i>	Z0	Valid	Valid	Input (E)CX words from port DX into ES:[(E)DI].
F3 6D	REP INS <i>m32, DX</i>	Z0	Valid	Valid	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 6D	REP INS <i>r/m32, DX</i>	Z0	Valid	N.E.	Input RCX default size from port DX into [RDI].
F3 A4	REP MOVS <i>m8, m8</i>	Z0	Valid	Valid	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A4	REP MOVS <i>m8, m8</i>	Z0	Valid	N.E.	Move RCX bytes from [RSI] to [RDI].
F3 A5	REP MOVS <i>m16, m16</i>	Z0	Valid	Valid	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS <i>m32, m32</i>	Z0	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A5	REP MOVS <i>m64, m64</i>	Z0	Valid	N.E.	Move RCX quadwords from [RSI] to [RDI].
F3 6E	REP OUTS <i>DX, r/m8</i>	Z0	Valid	Valid	Output (E)CX bytes from DS:[(E)SI] to port DX.
F3 REX.W 6E	REP OUTS <i>DX, r/m8*</i>	Z0	Valid	N.E.	Output RCX bytes from [RSI] to port DX.
F3 6F	REP OUTS <i>DX, r/m16</i>	Z0	Valid	Valid	Output (E)CX words from DS:[(E)SI] to port DX.
F3 6F	REP OUTS <i>DX, r/m32</i>	Z0	Valid	Valid	Output (E)CX doublewords from DS:[(E)SI] to port DX.
F3 REX.W 6F	REP OUTS <i>DX, r/m32</i>	Z0	Valid	N.E.	Output RCX default size from [RSI] to port DX.
F3 AC	REP LODS <i>AL</i>	Z0	Valid	Valid	Load (E)CX bytes from DS:[(E)SI] to AL.
F3 REX.W AC	REP LODS <i>AL</i>	Z0	Valid	N.E.	Load RCX bytes from [RSI] to AL.
F3 AD	REP LODS <i>AX</i>	Z0	Valid	Valid	Load (E)CX words from DS:[(E)SI] to AX.
F3 AD	REP LODS <i>EAX</i>	Z0	Valid	Valid	Load (E)CX doublewords from DS:[(E)SI] to EAX.
F3 REX.W AD	REP LODS <i>RAX</i>	Z0	Valid	N.E.	Load RCX quadwords from [RSI] to RAX.
F3 AA	REP STOS <i>m8</i>	Z0	Valid	Valid	Fill (E)CX bytes at ES:[(E)DI] with AL.
F3 REX.W AA	REP STOS <i>m8</i>	Z0	Valid	N.E.	Fill RCX bytes at [RDI] with AL.
F3 AB	REP STOS <i>m16</i>	Z0	Valid	Valid	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS <i>m32</i>	Z0	Valid	Valid	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
F3 REX.W AB	REP STOS <i>m64</i>	Z0	Valid	N.E.	Fill RCX quadwords at [RDI] with RAX.
F3 A6	REPE CMPS <i>m8, m8</i>	Z0	Valid	Valid	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A6	REPE CMPS <i>m8, m8</i>	Z0	Valid	N.E.	Find non-matching bytes in [RDI] and [RSI].
F3 A7	REPE CMPS <i>m16, m16</i>	Z0	Valid	Valid	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS <i>m32, m32</i>	Z0	Valid	Valid	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A7	REPE CMPS <i>m64, m64</i>	Z0	Valid	N.E.	Find non-matching quadwords in [RDI] and [RSI].
F3 AE	REPE SCAS <i>m8</i>	Z0	Valid	Valid	Find non-AL byte starting at ES:[(E)DI].
F3 REX.W AE	REPE SCAS <i>m8</i>	Z0	Valid	N.E.	Find non-AL byte starting at [RDI].
F3 AF	REPE SCAS <i>m16</i>	Z0	Valid	Valid	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS <i>m32</i>	Z0	Valid	Valid	Find non-EAX doubleword starting at ES:[(E)DI].

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
F3 REX.W AF	REPE SCAS <i>m64</i>	Z0	Valid	N.E.	Find non-RAX quadword starting at [RDI].
F2 A6	REPNE CMPS <i>m8, m8</i>	Z0	Valid	Valid	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A6	REPNE CMPS <i>m8, m8</i>	Z0	Valid	N.E.	Find matching bytes in [RDI] and [RSI].
F2 A7	REPNE CMPS <i>m16, m16</i>	Z0	Valid	Valid	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS <i>m32, m32</i>	Z0	Valid	Valid	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A7	REPNE CMPS <i>m64, m64</i>	Z0	Valid	N.E.	Find matching doublewords in [RDI] and [RSI].
F2 AE	REPNE SCAS <i>m8</i>	Z0	Valid	Valid	Find AL, starting at ES:[(E)DI].
F2 REX.W AE	REPNE SCAS <i>m8</i>	Z0	Valid	N.E.	Find AL, starting at [RDI].
F2 AF	REPNE SCAS <i>m16</i>	Z0	Valid	Valid	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS <i>m32</i>	Z0	Valid	Valid	Find EAX, starting at ES:[(E)DI].
F2 REX.W AF	REPNE SCAS <i>m64</i>	Z0	Valid	N.E.	Find RAX, starting at [RDI].

**NOTES:**

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The F3H prefix is defined for the following instructions and undefined for the rest:

- F3H as REP/REPE/REPZ for string and input/output instruction.
- F3H is a mandatory prefix for POPCNT, LZCNT, and ADOX.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. All of these repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0. See Table 4-17.

**Table 4-17. Repeat Prefixes**

Repeat Prefix	Termination Condition 1*	Termination Condition 2
REP	RCX or (E)CX = 0	None
REPE/REPZ	RCX or (E)CX = 0	ZF = 0
REPNE/REPNZ	RCX or (E)CX = 0	ZF = 1

**NOTES:**

\* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes.

The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

When the REPE/REPZ and REPNE/REPZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

In 64-bit mode, the operand size of the count register is associated with the address size attribute. Thus the default count register is RCX; REX.W has no effect on the address size and the count register. In 64-bit mode, if 67H is used to override address size attribute, the count register is ECX and any implicit source/destination operand will use the corresponding 32-bit index register. See the summary chart at the beginning of this section for encoding data and limits.

REP INS may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g. #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

## Operation

```

IF AddressSize = 16
  THEN
    Use CX for CountReg;
    Implicit Source/Dest operand for memory use of SI/DI;
  ELSE IF AddressSize = 64
    THEN Use RCX for CountReg;
    Implicit Source/Dest operand for memory use of RSI/RDI;
  ELSE
    Use ECX for CountReg;
    Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
  DO
    Service pending interrupts (if any);
    Execute associated string instruction;
    CountReg ← (CountReg - 1);
    IF CountReg = 0
      THEN exit WHILE loop; FI;
    IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
      or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
      THEN exit WHILE loop; FI;
  OD;

```

## Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

## Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

### 64-Bit Mode Exceptions

#GP(0)                    If the memory address is in a non-canonical form.

## RET—Return from Procedure

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C3	RET	Z0	Valid	Valid	Near return to calling procedure.
CB	RET	Z0	Valid	Valid	Far return to calling procedure.
C2 <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Near return to calling procedure and pop <i>imm16</i> bytes from stack.
CA <i>iw</i>	RET <i>imm16</i>	I	Valid	Valid	Far return to calling procedure and pop <i>imm16</i> bytes from stack.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA
I	<i>imm16</i>	NA	NA	NA

### Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure’s stack and the calling procedure’s stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.

**Operation**

(\* Near return \*)

IF instruction = near return

THEN;

IF OperandSize = 32

THEN

IF top 4 bytes of stack not within stack limits

THEN #SS(0); FI;

EIP  $\leftarrow$  Pop();

ELSE

IF OperandSize = 64

THEN

IF top 8 bytes of stack not within stack limits

THEN #SS(0); FI;

RIP  $\leftarrow$  Pop();

ELSE (\* OperandSize = 16 \*)

IF top 2 bytes of stack not within stack limits

THEN #SS(0); FI;

tempEIP  $\leftarrow$  Pop();tempEIP  $\leftarrow$  tempEIP AND 0000FFFFH;

IF tempEIP not within code segment limits

THEN #GP(0); FI;

EIP  $\leftarrow$  tempEIP;

FI;

FI;

IF instruction has immediate operand

THEN (\* Release parameters from stack \*)

IF StackAddressSize = 32

THEN

ESP  $\leftarrow$  ESP + SRC;

ELSE

IF StackAddressSize = 64

THEN

RSP  $\leftarrow$  RSP + SRC;

ELSE (\* StackAddressSize = 16 \*)

SP  $\leftarrow$  SP + SRC;

FI;

FI;

FI;

FI;

(\* Real-address mode or virtual-8086 mode \*)

IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return

THEN

IF OperandSize = 32

THEN

IF top 8 bytes of stack not within stack limits

THEN #SS(0); FI;

EIP  $\leftarrow$  Pop();CS  $\leftarrow$  Pop(); (\* 32-bit pop, high-order 16 bits discarded \*)

ELSE (\* OperandSize = 16 \*)

IF top 4 bytes of stack not within stack limits

THEN #SS(0); FI;

```

        tempEIP ← Pop();
        tempEIP ← tempEIP AND 0000FFFFH;
        IF tempEIP not within code segment limits
            THEN #GP(0); FI;
        EIP ← tempEIP;
        CS ← Pop(); (* 16-bit pop *)
    FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        SP ← SP + (SRC AND FFFFH);
    FI;
FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
                ELSE (* OperandSize = 16 *)
                    IF second word on stack is not within stack limits
                        THEN #SS(0); FI;
            FI;
        IF return code segment selector is NULL
            THEN #GP(0); FI;
        IF return code segment selector addresses descriptor beyond descriptor table limit
            THEN #GP(selector); FI;
        Obtain descriptor to which return code segment selector points from descriptor table;
        IF return code segment descriptor is not a code segment
            THEN #GP(selector); FI;
        IF return code segment selector RPL < CPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is conforming
            and return code segment DPL > return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is non-conforming and return code
            segment DPL ≠ return code segment selector RPL
            THEN #GP(selector); FI;
        IF return code segment descriptor is not present
            THEN #NP(selector); FI;
        IF return code segment selector RPL > CPL
            THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
            ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

RETURN-TO-SAME-PRIVILEGE-LEVEL:
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        FI;

```



```

    ELSE (* OperandSize = 16 *)
        EIP ← Pop();
        EIP ← EIP AND 0000FFFFH;
        CS ← Pop(); (* 16-bit pop *)
    FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                    SP ← SP + SRC;
            FI;
        FI;
    FI;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
    or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
        THEN #SS(0); FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    or stack segment is not a writable data segment
    or stack segment descriptor DPL ≠ RPL of the return code segment selector
        THEN #GP(selector); FI;
    IF stack segment not present
        THEN #SS(StackSegmentSelector); FI;
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    CPL ← ReturnCodeSegmentSelector(RPL);
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP + SRC;
                    FI;
                FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();

```

```

EIP ← EIP AND 0000FFFFH;
CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
CS(RPL) ← CPL;
IF instruction has immediate operand
    THEN (* Release parameters from called procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP ← SP + SRC;
        FI;
    FI;
tempESP ← Pop();
tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
ESP ← tempESP;
SS ← tempSS;
FI;

FOR each SegReg in (ES, FS, GS, and DS)
    DO
        tempDesc ← descriptor cache for SegReg (* hidden part of segment register *)
        IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
            THEN (* Segment register invalid *)
                SegmentSelector ← 0; (*Segment selector becomes null*)
        FI;
    OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE (* StackAddressSize = 16 *)
                SP ← SP + SRC;
        FI;
    FI;

(* IA-32e Mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
                IF first or second doubleword on stack is not in canonical space
                    THEN #SS(0); FI;
            ELSE
                IF OperandSize = 16
                    THEN
                        IF second word on stack is not within stack limits
                            THEN #SS(0); FI;
                        IF first or second word on stack is not in canonical space
                            THEN #SS(0); FI;
                    ELSE (* OperandSize = 64 *)

```

```

                IF first or second quadword on stack is not in canonical space
                  THEN #SS(0); FI;
            FI;
    FI;
    IF return code segment selector is NULL
      THEN GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
      THEN GP(selector); FI;
    IF return code segment selector addresses descriptor in non-canonical space
      THEN GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
      THEN #GP(selector); FI;
    IF return code segment descriptor has L-bit = 1 and D-bit = 1
      THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
      THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
      and return code segment DPL > return code segment selector RPL
      THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming
      and return code segment DPL ≠ return code segment selector RPL
      THEN #GP(selector); FI;
    IF return code segment descriptor is not present
      THEN #NP(selector); FI;
    IF return code segment selector RPL > CPL
      THEN GOTO IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL;
      ELSE GOTO IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
  FI;
FI;

```

#### IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL:

```

IF the return instruction pointer is not within the return code segment limit
  THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
  THEN #GP(0); FI;
IF OperandSize = 32
  THEN
    EIP ← Pop();
    CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
  ELSE
    IF OperandSize = 16
      THEN
        EIP ← Pop();
        EIP ← EIP AND 0000FFFFH;
        CS ← Pop(); (* 16-bit pop *)
      ELSE (* OperandSize = 64 *)
        RIP ← Pop();
        CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
    FI;
  FI;
FI;
IF instruction has immediate operand
  THEN (* Release parameters from stack *)
    IF StackAddressSize = 32

```

```

    THEN
        ESP ← ESP + SRC;
    ELSE
        IF StackAddressSize = 16
            THEN
                SP ← SP + SRC;
            ELSE (* StackAddressSize = 64 *)
                RSP ← RSP + SRC;
        FI;
    FI;
FI;

IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL:
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
    FI;
IF return stack segment descriptor is not within descriptor table limits
    THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
    THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;

```

```

        ELSE
            IF StackAddressSize = 16
                THEN
                    SP ← SP + SRC;
                ELSE (* StackAddressSize = 64 *)
                    RSP ← RSP + SRC;
            FI;
        FI;
    FI;
    tempESP ← Pop();
    tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
    ESP ← tempESP;
    SS ← tempSS;
ELSE
    IF OperandSize = 16
        THEN
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP + SRC;
                        ELSE
                            IF StackAddressSize = 16
                                THEN
                                    SP ← SP + SRC;
                                ELSE (* StackAddressSize = 64 *)
                                    RSP ← RSP + SRC;
                            FI;
                        FI;
                    FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE (* OperandSize = 64 *)
            RIP ← Pop();
            CS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    RSP ← RSP + SRC;
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        FI;
    FI;
FOR each of segment register (ES, FS, GS, and DS)

```

```

DO
    IF segment register points to data or non-conforming code segment
    and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
    THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
FI;
OD;

IF instruction has immediate operand
THEN (* Release parameters from calling procedure's stack *)
    IF StackAddressSize = 32
    THEN
        ESP ← ESP + SRC;
    ELSE
        IF StackAddressSize = 16
        THEN
            SP ← SP + SRC;
        ELSE (* StackAddressSize = 64 *)
            RSP ← RSP + SRC;
        FI;
    FI;
FI;

```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL.
	If the return instruction pointer is not within the return code segment limit
#GP(selector)	If the RPL of the return code segment selector is less than the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.
	If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector
	If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

### Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

**Virtual-8086 Mode Exceptions**

#GP(0)	If the return instruction pointer is not within the return code segment limit
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

**Compatibility Mode Exceptions**

Same as 64-bit mode exceptions.

**64-Bit Mode Exceptions**

#GP(0)	<p>If the return instruction pointer is non-canonical.</p> <p>If the return instruction pointer is not within the return code segment limit.</p> <p>If the stack segment selector is NULL going back to compatibility mode.</p> <p>If the stack segment selector is NULL going back to CPL3 64-bit mode.</p> <p>If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.</p> <p>If the return code segment selector is NULL.</p>
#GP(selector)	<p>If the proposed segment descriptor for a code segment does not indicate it is a code segment.</p> <p>If the proposed new code segment descriptor has both the D-bit and L-bit set.</p> <p>If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.</p> <p>If CPL is greater than the RPL of the code segment selector.</p> <p>If the DPL of a conforming-code segment is greater than the return code segment selector RPL.</p> <p>If a segment selector index is outside its descriptor table limits.</p> <p>If a segment descriptor memory address is non-canonical.</p> <p>If the stack segment is not a writable data segment.</p> <p>If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.</p> <p>If the stack segment selector RPL is not equal to the RPL of the return code segment selector.</p>
#SS(0)	<p>If an attempt to pop a value off the stack violates the SS limit.</p> <p>If an attempt to pop a value off the stack causes a non-canonical address to be referenced.</p>
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## RORX — Rotate Right Logical Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F3A.W0 F0 /r ib RORX r32, r/m32, imm8	RMI	V/V	BMI2	Rotate 32-bit <i>r/m32</i> right <i>imm8</i> times without affecting arithmetic flags.
VEX.LZ.F2.0F3A.W1 F0 /r ib RORX r64, r/m64, imm8	RMI	V/N.E.	BMI2	Rotate 64-bit <i>r/m64</i> right <i>imm8</i> times without affecting arithmetic flags.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Rotates the bits of second operand right by the count value specified in *imm8* without affecting arithmetic flags. The RORX instruction does not read or write the arithmetic flags.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

### Operation

```

IF (OperandSize = 32)
    y ← imm8 AND 1FH;
    DEST ← (SRC >> y) | (SRC << (32-y));
ELSEIF (OperandSize = 64)
    y ← imm8 AND 3FH;
    DEST ← (SRC >> y) | (SRC << (64-y));
ENDIF

```

### Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally  
 #UD If VEX.W = 1.



## ROUNDPD — Round Packed Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 09 /r ib ROUNDPD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Round packed double precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.128.66.0F3A.WIG 09 /r ib VROUNDPD <i>xmm1</i> , <i>xmm2/m128</i> , <i>imm8</i>	RMI	V/V	AVX	Round packed double-precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.256.66.0F3A.WIG 09 /r ib VROUNDPD <i>ymm1</i> , <i>ymm2/m256</i> , <i>imm8</i>	RMI	V/V	AVX	Round packed double-precision floating-point values in <i>ymm2/m256</i> and place the result in <i>ymm1</i> . The rounding mode is determined by <i>imm8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

### Description

Round the 2 double-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a double-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

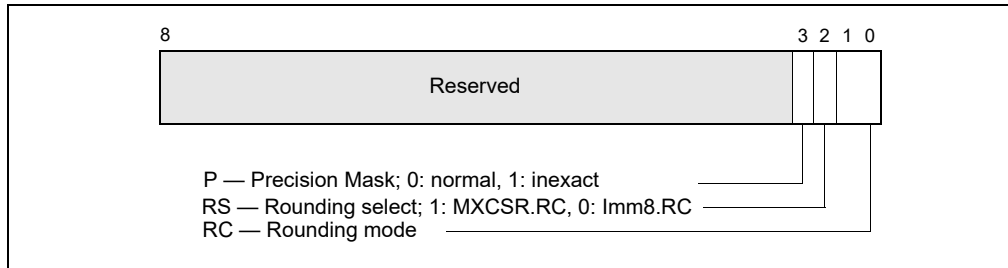


Figure 4-24. Bit Control Fields of Immediate Byte for ROUNDxx Instruction

Table 4-18. Rounding Modes and Encoding of Rounding Control (RC) Field

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the integer value with the least-significant bit of zero).
Round down (toward $-\infty$ )	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward $+\infty$ )	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

### Operation

```

IF (imm[2] = '1)
    THEN // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_M(SRC[127:64]);
    ELSE // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_Imm(SRC[127:64]);
FI

```

#### ROUNDPD (128-bit Legacy SSE version)

```

DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)

```

#### VROUNDPD (VEX.128 encoded version)

```

DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64], ROUND_CONTROL)
DEST[MAXVL-1:128] ← 0

```

#### VROUNDPD (VEX.256 encoded version)

```

DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64], ROUND_CONTROL)
DEST[191:128] ← RoundToInteger(SRC[191:128], ROUND_CONTROL)
DEST[255:192] ← RoundToInteger(SRC[255:192], ROUND_CONTROL)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

__m128 _mm_round_pd(__m128d s1, int iRoundMode);
__m128 _mm_floor_pd(__m128d s1);
__m128 _mm_ceil_pd(__m128d s1)
__m256 _mm256_round_pd(__m256d s1, int iRoundMode);
__m256 _mm256_floor_pd(__m256d s1);
__m256 _mm256_ceil_pd(__m256d s1)

```

**SIMD Floating-Point Exceptions**

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0'; if imm[3] = '1', then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPD.

**Other Exceptions**

See Exceptions Type 2; additionally

#UD                      If VEX.vvvv ≠ 1111B.

## ROUNDPS — Round Packed Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 08 /r ib ROUNDPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	SSE4_1	Round packed single precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.128.66.0F3A.WIG 08 /r ib VROUNDPS <i>xmm1, xmm2/m128, imm8</i>	RMI	V/V	AVX	Round packed single-precision floating-point values in <i>xmm2/m128</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.256.66.0F3A.WIG 08 /r ib VROUNDPS <i>ymm1, ymm2/m256, imm8</i>	RMI	V/V	AVX	Round packed single-precision floating-point values in <i>ymm2/m256</i> and place the result in <i>ymm1</i> . The rounding mode is determined by <i>imm8</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

### Description

Round the 4 single-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

```

IF (imm[2] = '1')
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_M(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_M(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_M(SRC[127:96]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_Imm(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_Imm(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;

```

**ROUNDPS(128-bit Legacy SSE version)**

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)

```

**VROUNDPS (VEX.128 encoded version)**

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[MAXVL-1:128] ← 0

```

**VROUNDPS (VEX.256 encoded version)**

```

DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96], ROUND_CONTROL)
DEST[159:128] ← RoundToInteger(SRC[159:128], ROUND_CONTROL)
DEST[191:160] ← RoundToInteger(SRC[191:160], ROUND_CONTROL)
DEST[223:192] ← RoundToInteger(SRC[223:192], ROUND_CONTROL)
DEST[255:224] ← RoundToInteger(SRC[255:224], ROUND_CONTROL)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

__m128 _mm_round_ps(__m128 s1, int iRoundMode);
__m128 _mm_floor_ps(__m128 s1);
__m128 _mm_ceil_ps(__m128 s1)
__m256 _mm256_round_ps(__m256 s1, int iRoundMode);
__m256 _mm256_floor_ps(__m256 s1);
__m256 _mm256_ceil_ps(__m256 s1)

```

### SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPS.

### Other Exceptions

See Exceptions Type 2; additionally

#UD                      If VEX.vvvv ≠ 1111B.

## ROUNDSD — Round Scalar Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0B /r ib ROUNDSD <i>xmm1</i> , <i>xmm2/m64</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Round the low packed double precision floating-point value in <i>xmm2/m64</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.NDS.LIG.66.0F3A.WIG 0B /r ib VROUNDSD <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m64</i> , <i>imm8</i>	RVMI	V/V	AVX	Round the low packed double precision floating-point value in <i>xmm3/m64</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> . Upper packed double precision floating-point value (bits[127:64]) from <i>xmm2</i> is copied to <i>xmm1</i> [127:64].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Round the DP FP value in the lower qword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a double-precision floating-point input to an integer value and returns the integer result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

### Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged ;
```

#### ROUNDSD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[MAXVL-1:64] (Unmodified)
```

**VROUNDSD (VEX.128 encoded version)**

DEST[63:0]  $\leftarrow$  RoundToInteger(SRC2[63:0], ROUND\_CONTROL)

DEST[127:64]  $\leftarrow$  SRC1[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

```
ROUNDSD:    __m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);
             __m128d mm_floor_sd(__m128d dst, __m128d s1);
             __m128d mm_ceil_sd(__m128d dst, __m128d s1);
```

**SIMD Floating-Point Exceptions**

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0'; if imm[3] = '1', then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSD.

**Other Exceptions**

See Exceptions Type 3.



## ROUNDSS — Round Scalar Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0A /r ib ROUNDSS <i>xmm1</i> , <i>xmm2/m32</i> , <i>imm8</i>	RMI	V/V	SSE4_1	Round the low packed single precision floating-point value in <i>xmm2/m32</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> .
VEX.NDS.LIG.66.0F3A.WIG 0A /r ib VROUNDSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i> , <i>imm8</i>	RVMI	V/V	AVX	Round the low packed single precision floating-point value in <i>xmm3/m32</i> and place the result in <i>xmm1</i> . The rounding mode is determined by <i>imm8</i> . Also, upper packed single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

### Description

Round the single-precision floating-point value in the lowest dword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a single-precision floating-point input to an integer value and returns the result as a single-precision floating-point value in the lowest position. The upper three single-precision floating-point values in the destination are retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

### Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
            DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
    ELSE    // rounding mode is determined by IMM8.RC
            DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged ;
```

#### ROUNDSS (128-bit Legacy SSE version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[MAXVL-1:32] (Unmodified)
```

**VROUNDSS (VEX.128 encoded version)**

DEST[31:0]  $\leftarrow$  RoundToInteger(SRC2[31:0], ROUND\_CONTROL)

DEST[127:32]  $\leftarrow$  SRC1[127:32]

DEST[MAXVL-1:128]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

ROUNDSS:     \_\_m128 mm\_round\_ss(\_\_m128 dst, \_\_m128 s1, int iRoundMode);  
               \_\_m128 mm\_floor\_ss(\_\_m128 dst, \_\_m128 s1);  
               \_\_m128 mm\_ceil\_ss(\_\_m128 dst, \_\_m128 s1);

**SIMD Floating-Point Exceptions**

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0'; if imm[3] = '1', then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSS.

**Other Exceptions**

See Exceptions Type 3.

## RSM—Resume from System Management Mode

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF AA	RSM	Z0	Valid	Valid	Resume operation of interrupted program.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486™ processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

The SMM state map used by RSM supports resuming processor context for non-64-bit modes and 64-bit mode.

See Chapter 34, "System Management Mode," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about SMM and the behavior of the RSM instruction.

### Operation

ReturnFromSMM;

IF (IA-32e mode supported) or (CPUID DisplayFamily\_DisplayModel = 06H\_0CH )

THEN

ProcessorState ← Restore(SMMDump(IA-32e SMM STATE MAP));

Else

ProcessorState ← Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));

FI

### Flags Affected

All.

### Protected Mode Exceptions

#UD If an attempt is made to execute this instruction when the processor is not in SMM.  
If the LOCK prefix is used.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 52 /r RSQRTPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	SSE	Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in <i>xmm2/m128</i> and stores the results in <i>xmm1</i> .
VEX.128.OF.WIG 52 /r VRSQRTPS <i>xmm1</i> , <i>xmm2/m128</i>	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in <i>xmm2/mem</i> and stores the results in <i>xmm1</i> .
VEX.256.OF.WIG 52 /r VRSQRTPS <i>ymm1</i> , <i>ymm2/m256</i>	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in <i>ymm2/mem</i> and stores the results in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four packed single-precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than -0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

## Operation

### RSQRTPS (128-bit Legacy SSE version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC}[31:0]))$   
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[63:32]))$   
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[95:64]))$   
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[127:96]))$   
 $\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}$

### VRSQRTPS (VEX.128 encoded version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC}[31:0]))$   
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[63:32]))$   
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[95:64]))$   
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[127:96]))$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$

### VRSQRTPS (VEX.256 encoded version)

$\text{DEST}[31:0] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC}[31:0]))$   
 $\text{DEST}[63:32] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[63:32]))$   
 $\text{DEST}[95:64] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC1}[95:64]))$   
 $\text{DEST}[127:96] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[127:96]))$   
 $\text{DEST}[159:128] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[159:128]))$   
 $\text{DEST}[191:160] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[191:160]))$   
 $\text{DEST}[223:192] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[223:192]))$   
 $\text{DEST}[255:224] \leftarrow \text{APPROXIMATE}(1/\text{SQRT}(\text{SRC2}[255:224]))$

## Intel C/C++ Compiler Intrinsic Equivalent

RSQRTPS: `__m128 _mm_rsqrtps(__m128 a)`  
 RSQRTPS: `__m256 _mm256_rsqrtps (__m256 a);`

## SIMD Floating-Point Exceptions

None.

## Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.vvvv  $\neq$  1111B.

## RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 52 /r RSQRTSS <i>xmm1</i> , <i>xmm2/m32</i>	RM	V/V	SSE	Computes the approximate reciprocal of the square root of the low single-precision floating-point value in <i>xmm2/m32</i> and stores the results in <i>xmm1</i> .
VEX.NDS.LIG.F3.0F.WIG 52 /r VRSQRTSS <i>xmm1</i> , <i>xmm2</i> , <i>xmm3/m32</i>	RVM	V/V	AVX	Computes the approximate reciprocal of the square root of the low single precision floating-point value in <i>xmm3/m32</i> and stores the results in <i>xmm1</i> . Also, upper single precision floating-point values (bits[127:32]) from <i>xmm2</i> are copied to <i>xmm1</i> [127:32].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the source operand (second operand) stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than -0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

### Operation

#### RSQRTSS (128-bit Legacy SSE version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))

DEST[MAXVL-1:32] (Unmodified)

#### VRSQRTSS (VEX.128 encoded version)

DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))

DEST[127:32] ← SRC1[127:32]

DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

RSQRTSS: `__m128 _mm_rsqrt_ss(__m128 a)`

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type 5.



## SAHF—Store AH into Flags

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9E	SAHF	Z0	Invalid*	Valid	Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register.

### NOTES:

\* Valid in specific steppings. See Description section.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the “Operation” section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

### Operation

```
IF IA-64 Mode
  THEN
    IF CPUID.80000001H:ECX[0] = 1;
      THEN
        RFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
      ELSE
        #UD;
    FI
  ELSE
    EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
FI;
```

### Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

### Protected Mode Exceptions

None.

### Real-Address Mode Exceptions

None.

### Virtual-8086 Mode Exceptions

None.

### Compatibility Mode Exceptions

None.

**64-Bit Mode Exceptions**

#UD                    If CPUID.80000001H.ECX[0] = 0.  
                      If the LOCK prefix is used.

## SAL/SAR/SHL/SHR—Shift

Opcode***	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
D0 /4	SAL <i>r/m8</i> , 1	M1	Valid	Valid	Multiply <i>r/m8</i> by 2, once.
REX + D0 /4	SAL <i>r/m8**</i> , 1	M1	Valid	N.E.	Multiply <i>r/m8</i> by 2, once.
D2 /4	SAL <i>r/m8</i> , CL	MC	Valid	Valid	Multiply <i>r/m8</i> by 2, CL times.
REX + D2 /4	SAL <i>r/m8**</i> , CL	MC	Valid	N.E.	Multiply <i>r/m8</i> by 2, CL times.
C0 /4 <i>ib</i>	SAL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
REX + C0 /4 <i>ib</i>	SAL <i>r/m8**</i> , <i>imm8</i>	MI	Valid	N.E.	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /4	SAL <i>r/m16</i> , 1	M1	Valid	Valid	Multiply <i>r/m16</i> by 2, once.
D3 /4	SAL <i>r/m16</i> , CL	MC	Valid	Valid	Multiply <i>r/m16</i> by 2, CL times.
C1 /4 <i>ib</i>	SAL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /4	SAL <i>r/m32</i> , 1	M1	Valid	Valid	Multiply <i>r/m32</i> by 2, once.
REX.W + D1 /4	SAL <i>r/m64</i> , 1	M1	Valid	N.E.	Multiply <i>r/m64</i> by 2, once.
D3 /4	SAL <i>r/m32</i> , CL	MC	Valid	Valid	Multiply <i>r/m32</i> by 2, CL times.
REX.W + D3 /4	SAL <i>r/m64</i> , CL	MC	Valid	N.E.	Multiply <i>r/m64</i> by 2, CL times.
C1 /4 <i>ib</i>	SAL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /4 <i>ib</i>	SAL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Multiply <i>r/m64</i> by 2, <i>imm8</i> times.
D0 /7	SAR <i>r/m8</i> , 1	M1	Valid	Valid	Signed divide* <i>r/m8</i> by 2, once.
REX + D0 /7	SAR <i>r/m8**</i> , 1	M1	Valid	N.E.	Signed divide* <i>r/m8</i> by 2, once.
D2 /7	SAR <i>r/m8</i> , CL	MC	Valid	Valid	Signed divide* <i>r/m8</i> by 2, CL times.
REX + D2 /7	SAR <i>r/m8**</i> , CL	MC	Valid	N.E.	Signed divide* <i>r/m8</i> by 2, CL times.
C0 /7 <i>ib</i>	SAR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Signed divide* <i>r/m8</i> by 2, <i>imm8</i> time.
REX + C0 /7 <i>ib</i>	SAR <i>r/m8**</i> , <i>imm8</i>	MI	Valid	N.E.	Signed divide* <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /7	SAR <i>r/m16</i> , 1	M1	Valid	Valid	Signed divide* <i>r/m16</i> by 2, once.
D3 /7	SAR <i>r/m16</i> , CL	MC	Valid	Valid	Signed divide* <i>r/m16</i> by 2, CL times.
C1 /7 <i>ib</i>	SAR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Signed divide* <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /7	SAR <i>r/m32</i> , 1	M1	Valid	Valid	Signed divide* <i>r/m32</i> by 2, once.
REX.W + D1 /7	SAR <i>r/m64</i> , 1	M1	Valid	N.E.	Signed divide* <i>r/m64</i> by 2, once.
D3 /7	SAR <i>r/m32</i> , CL	MC	Valid	Valid	Signed divide* <i>r/m32</i> by 2, CL times.
REX.W + D3 /7	SAR <i>r/m64</i> , CL	MC	Valid	N.E.	Signed divide* <i>r/m64</i> by 2, CL times.
C1 /7 <i>ib</i>	SAR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Signed divide* <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /7 <i>ib</i>	SAR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Signed divide* <i>r/m64</i> by 2, <i>imm8</i> times
D0 /4	SHL <i>r/m8</i> , 1	M1	Valid	Valid	Multiply <i>r/m8</i> by 2, once.
REX + D0 /4	SHL <i>r/m8**</i> , 1	M1	Valid	N.E.	Multiply <i>r/m8</i> by 2, once.
D2 /4	SHL <i>r/m8</i> , CL	MC	Valid	Valid	Multiply <i>r/m8</i> by 2, CL times.
REX + D2 /4	SHL <i>r/m8**</i> , CL	MC	Valid	N.E.	Multiply <i>r/m8</i> by 2, CL times.
C0 /4 <i>ib</i>	SHL <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
REX + C0 /4 <i>ib</i>	SHL <i>r/m8**</i> , <i>imm8</i>	MI	Valid	N.E.	Multiply <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /4	SHL <i>r/m16</i> , 1	M1	Valid	Valid	Multiply <i>r/m16</i> by 2, once.
D3 /4	SHL <i>r/m16</i> , CL	MC	Valid	Valid	Multiply <i>r/m16</i> by 2, CL times.
C1 /4 <i>ib</i>	SHL <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /4	SHL <i>r/m32</i> , 1	M1	Valid	Valid	Multiply <i>r/m32</i> by 2, once.

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
REX.W + D1 /4	SHL <i>r/m64</i> , 1	M1	Valid	N.E.	Multiply <i>r/m64</i> by 2, once.
D3 /4	SHL <i>r/m32</i> , CL	MC	Valid	Valid	Multiply <i>r/m32</i> by 2, CL times.
REX.W + D3 /4	SHL <i>r/m64</i> , CL	MC	Valid	N.E.	Multiply <i>r/m64</i> by 2, CL times.
C1 /4 <i>ib</i>	SHL <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Multiply <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /4 <i>ib</i>	SHL <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Multiply <i>r/m64</i> by 2, <i>imm8</i> times.
D0 /5	SHR <i>r/m8</i> , 1	M1	Valid	Valid	Unsigned divide <i>r/m8</i> by 2, once.
REX + D0 /5	SHR <i>r/m8**</i> , 1	M1	Valid	N.E.	Unsigned divide <i>r/m8</i> by 2, once.
D2 /5	SHR <i>r/m8</i> , CL	MC	Valid	Valid	Unsigned divide <i>r/m8</i> by 2, CL times.
REX + D2 /5	SHR <i>r/m8**</i> , CL	MC	Valid	N.E.	Unsigned divide <i>r/m8</i> by 2, CL times.
C0 /5 <i>ib</i>	SHR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Unsigned divide <i>r/m8</i> by 2, <i>imm8</i> times.
REX + C0 /5 <i>ib</i>	SHR <i>r/m8**</i> , <i>imm8</i>	MI	Valid	N.E.	Unsigned divide <i>r/m8</i> by 2, <i>imm8</i> times.
D1 /5	SHR <i>r/m16</i> , 1	M1	Valid	Valid	Unsigned divide <i>r/m16</i> by 2, once.
D3 /5	SHR <i>r/m16</i> , CL	MC	Valid	Valid	Unsigned divide <i>r/m16</i> by 2, CL times.
C1 /5 <i>ib</i>	SHR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Unsigned divide <i>r/m16</i> by 2, <i>imm8</i> times.
D1 /5	SHR <i>r/m32</i> , 1	M1	Valid	Valid	Unsigned divide <i>r/m32</i> by 2, once.
REX.W + D1 /5	SHR <i>r/m64</i> , 1	M1	Valid	N.E.	Unsigned divide <i>r/m64</i> by 2, once.
D3 /5	SHR <i>r/m32</i> , CL	MC	Valid	Valid	Unsigned divide <i>r/m32</i> by 2, CL times.
REX.W + D3 /5	SHR <i>r/m64</i> , CL	MC	Valid	N.E.	Unsigned divide <i>r/m64</i> by 2, CL times.
C1 /5 <i>ib</i>	SHR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Unsigned divide <i>r/m32</i> by 2, <i>imm8</i> times.
REX.W + C1 /5 <i>ib</i>	SHR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Unsigned divide <i>r/m64</i> by 2, <i>imm8</i> times.

**NOTES:**

\* Not the same form of division as IDIV; rounding is toward negative infinity.

\*\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

\*\*\*See IA-32 Architecture Compatibility section below.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m ( <i>r</i> , <i>w</i> )	1	NA	NA
MC	ModRM:r/m ( <i>r</i> , <i>w</i> )	CL	NA	NA
MI	ModRM:r/m ( <i>r</i> , <i>w</i> )	<i>imm8</i>	NA	NA

**Description**

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W is used). The count range is limited to 0 to 31 (or 63 if 64-bit mode and REX.W is used). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction's default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

## IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

## Operation

IF 64-Bit Mode and using REX.W

THEN

countMASK  $\leftarrow$  3FH;

ELSE

countMASK  $\leftarrow$  1FH;

FI

tempCOUNT  $\leftarrow$  (COUNT AND countMASK);

tempDEST  $\leftarrow$  DEST;

WHILE (tempCOUNT  $\neq$  0)

DO

IF instruction is SAL or SHL

THEN

CF  $\leftarrow$  MSB(DEST);

ELSE (\* Instruction is SAR or SHR \*)

CF  $\leftarrow$  LSB(DEST);

FI;

IF instruction is SAL or SHL

THEN

DEST  $\leftarrow$  DEST \* 2;

ELSE

IF instruction is SAR

```

        THEN
            DEST ← DEST / 2; (* Signed divide, rounding toward negative infinity *)
        ELSE (* Instruction is SHR *)
            DEST ← DEST / 2; (* Unsigned divide *)
    FI;
FI;
tempCOUNT ← tempCOUNT - 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* Instruction is SHR *)
                        OF ← MSB(tempDEST);
                FI;
            FI;
        ELSE IF (COUNT AND countMASK) = 0
            THEN
                All flags unchanged;
            ELSE (* COUNT not 1 or 0 *)
                OF ← undefined;
        FI;
FI;

```

### Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see “Description” above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## SARX/SHLX/SHRX — Shift Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F3.0F38.W0 F7 /r SARX <i>r32a</i> , <i>r/m32</i> , <i>r32b</i>	RMV	V/V	BMI2	Shift <i>r/m32</i> arithmetically right with count specified in <i>r32b</i> .
VEX.NDS.LZ.66.0F38.W0 F7 /r SHLX <i>r32a</i> , <i>r/m32</i> , <i>r32b</i>	RMV	V/V	BMI2	Shift <i>r/m32</i> logically left with count specified in <i>r32b</i> .
VEX.NDS.LZ.F2.0F38.W0 F7 /r SHRX <i>r32a</i> , <i>r/m32</i> , <i>r32b</i>	RMV	V/V	BMI2	Shift <i>r/m32</i> logically right with count specified in <i>r32b</i> .
VEX.NDS.LZ.F3.0F38.W1 F7 /r SARX <i>r64a</i> , <i>r/m64</i> , <i>r64b</i>	RMV	V/N.E.	BMI2	Shift <i>r/m64</i> arithmetically right with count specified in <i>r64b</i> .
VEX.NDS.LZ.66.0F38.W1 F7 /r SHLX <i>r64a</i> , <i>r/m64</i> , <i>r64b</i>	RMV	V/N.E.	BMI2	Shift <i>r/m64</i> logically left with count specified in <i>r64b</i> .
VEX.NDS.LZ.F2.0F38.W1 F7 /r SHRX <i>r64a</i> , <i>r/m64</i> , <i>r64b</i>	RMV	V/N.E.	BMI2	Shift <i>r/m64</i> logically right with count specified in <i>r64b</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

### Description

Shifts the bits of the first source operand (the second operand) to the left or right by a COUNT value specified in the second source operand (the third operand). The result is written to the destination operand (the first operand).

The shift arithmetic right (SARX) and shift logical right (SHRX) instructions shift the bits of the destination operand to the right (toward less significant bit locations), SARX keeps and propagates the most significant bit (sign bit) while shifting.

The logical shift left (SHLX) shifts the bits of the destination operand to the left (toward more significant bit locations).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

If the value specified in the first source operand exceeds OperandSize - 1, the COUNT value is masked.

SARX,SHRX, and SHLX instructions do not update flags.

### Operation

```
TEMP ← SRC1;
IF VEX.W1 and CS.L = 1
THEN
    countMASK ← 3FH;
ELSE
    countMASK ← 1FH;
FI
COUNT ← (SRC2 AND countMASK)
```

```
DEST[OperandSize - 1] = TEMP[OperandSize - 1];
DO WHILE (COUNT ≠ 0)
    IF instruction is SHLX
    THEN
        DEST[] ← DEST * 2;
```



```

    ELSE IF instruction is SHRX
      THEN
        DEST[] ← DEST /2; //unsigned divide
      ELSE
        // SARX
        DEST[] ← DEST /2; // signed divide, round toward negative infinity
    FI;
    COUNT ← COUNT - 1;
  OD

```

**Flags Affected**

None.

**Intel C/C++ Compiler Intrinsic Equivalent**

Auto-generated from high-level language.

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD                      If VEX.W = 1.

## SBB—Integer Subtraction with Borrow

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
1C <i>ib</i>	SBB AL, <i>imm8</i>	I	Valid	Valid	Subtract with borrow <i>imm8</i> from AL.
1D <i>iw</i>	SBB AX, <i>imm16</i>	I	Valid	Valid	Subtract with borrow <i>imm16</i> from AX.
1D <i>id</i>	SBB EAX, <i>imm32</i>	I	Valid	Valid	Subtract with borrow <i>imm32</i> from EAX.
REX.W + 1D <i>id</i>	SBB RAX, <i>imm32</i>	I	Valid	N.E.	Subtract with borrow sign-extended <i>imm32</i> to 64-bits from RAX.
80 /3 <i>ib</i>	SBB <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Subtract with borrow <i>imm8</i> from <i>r/m8</i> .
REX + 80 /3 <i>ib</i>	SBB <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Subtract with borrow <i>imm8</i> from <i>r/m8</i> .
81 /3 <i>iw</i>	SBB <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Subtract with borrow <i>imm16</i> from <i>r/m16</i> .
81 /3 <i>id</i>	SBB <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Subtract with borrow <i>imm32</i> from <i>r/m32</i> .
REX.W + 81 /3 <i>id</i>	SBB <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Subtract with borrow sign-extended <i>imm32</i> to 64-bits from <i>r/m64</i> .
83 /3 <i>ib</i>	SBB <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m16</i> .
83 /3 <i>ib</i>	SBB <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m32</i> .
REX.W + 83 /3 <i>ib</i>	SBB <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Subtract with borrow sign-extended <i>imm8</i> from <i>r/m64</i> .
18 /r	SBB <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Subtract with borrow <i>r8</i> from <i>r/m8</i> .
REX + 18 /r	SBB <i>r/m8*</i> , <i>r8</i>	MR	Valid	N.E.	Subtract with borrow <i>r8</i> from <i>r/m8</i> .
19 /r	SBB <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Subtract with borrow <i>r16</i> from <i>r/m16</i> .
19 /r	SBB <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Subtract with borrow <i>r32</i> from <i>r/m32</i> .
REX.W + 19 /r	SBB <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Subtract with borrow <i>r64</i> from <i>r/m64</i> .
1A /r	SBB <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Subtract with borrow <i>r/m8</i> from <i>r8</i> .
REX + 1A /r	SBB <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Subtract with borrow <i>r/m8</i> from <i>r8</i> .
1B /r	SBB <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Subtract with borrow <i>r/m16</i> from <i>r16</i> .
1B /r	SBB <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Subtract with borrow <i>r/m32</i> from <i>r32</i> .
REX.W + 1B /r	SBB <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Subtract with borrow <i>r/m64</i> from <i>r64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	<i>imm8/16/32</i>	NA	NA
MI	ModRM: <i>r/m</i> ( <i>w</i> )	<i>imm8/16/32</i>	NA	NA
MR	ModRM: <i>r/m</i> ( <i>w</i> )	ModRM: <i>reg</i> ( <i>r</i> )	NA	NA
RM	ModRM: <i>reg</i> ( <i>w</i> )	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA

## Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

## Operation

$DEST \leftarrow (DEST - (SRC + CF));$

## Intel C/C++ Compiler Intrinsic Equivalent

SBB: `extern unsigned char _subborrow_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *diff_out);`

SBB: `extern unsigned char _subborrow_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *diff_out);`

SBB: `extern unsigned char _subborrow_u32(unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *diff_out);`

SBB: `extern unsigned char _subborrow_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *diff_out);`

## Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## SCAS/SCASB/SCASW/SCASD—Scan String

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
AE	SCAS <i>m8</i>	ZO	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI, then set status flags.*
AF	SCAS <i>m16</i>	ZO	Valid	Valid	Compare AX with word at ES:(E)DI or RDI, then set status flags.*
AF	SCAS <i>m32</i>	ZO	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.*
REX.W + AF	SCAS <i>m64</i>	ZO	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.
AE	SCASB	ZO	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI then set status flags.*
AF	SCASW	ZO	Valid	Valid	Compare AX with word at ES:(E)DI or RDI then set status flags.*
AF	SCASD	ZO	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.*
REX.W + AF	SCASQ	ZO	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.

## NOTES:

\* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

## Description

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES:(E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed. The explicit-operand form and the no-operands form. The explicit-operand form (specified using the SCAS mnemonic) allows a memory operand to be specified explicitly. The memory operand must be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (AL register for byte comparisons, AX for word comparisons, EAX for doubleword comparisons). The explicit-operand form is provided to allow documentation. Note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword) but it does not have to specify the correct location. The location is always specified by ES:(E)DI.

The no-operands form of the instruction uses a short form of SCAS. Again, ES:(E)DI is assumed to be the memory operand and AL, AX, or EAX is assumed to be the register operand. The size of operands is selected by the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented. The register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REP prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes

some action based on the setting of status flags. See “REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix” in this chapter for a description of the REP prefix.

In 64-bit mode, the instruction’s default address size is 64-bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The 64-bit no-operand mnemonic is SCASQ. Address of the memory operand is specified in either RDI or EDI, and AL/AX/EAX/RAX may be used as the register operand. After a comparison, the destination register is incremented or decremented by the current operand size (depending on the value of the DF flag). See the summary chart at the beginning of this section for encoding data and limits.

## Operation

Non-64-bit Mode:

```
IF (Byte comparison)
    THEN
        temp ← AL – SRC;
        SetStatusFlags(temp);
        THEN IF DF = 0
            THEN (E)DI ← (E)DI + 1;
            ELSE (E)DI ← (E)DI – 1; FI;
    ELSE IF (Word comparison)
        THEN
            temp ← AX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI ← (E)DI + 2;
                ELSE (E)DI ← (E)DI – 2; FI;
        FI;
    ELSE IF (Doubleword comparison)
        THEN
            temp ← EAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI ← (E)DI + 4;
                ELSE (E)DI ← (E)DI – 4; FI;
        FI;
    FI;
```

64-bit Mode:

```
IF (Byte comparison)
    THEN
        temp ← AL – SRC;
        SetStatusFlags(temp);
        THEN IF DF = 0
            THEN (R)E)DI ← (R)E)DI + 1;
            ELSE (R)E)DI ← (R)E)DI – 1; FI;
    ELSE IF (Word comparison)
        THEN
            temp ← AX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R)E)DI ← (R)E)DI + 2;
                ELSE (R)E)DI ← (R)E)DI – 2; FI;
        FI;
    FI;
```

```

ELSE IF (Doubleword comparison)
  THEN
    temp ← EAX - SRC;
    SetStatusFlags(temp);
    IF DF = 0
      THEN (R|E)DI ← (R|E)DI + 4;
      ELSE (R|E)DI ← (R|E)DI - 4; FI;
    FI;
ELSE IF (Quadword comparison using REX.W )
  THEN
    temp ← RAX - SRC;
    SetStatusFlags(temp);
    IF DF = 0
      THEN (R|E)DI ← (R|E)DI + 8;
      ELSE (R|E)DI ← (R|E)DI - 8;
    FI;
  FI;
F

```

### Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment. If the ES register contains a NULL segment selector. If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.



## SETcc—Set Byte on Condition

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 97	SETA <i>r/m8</i>	M	Valid	Valid	Set byte if above (CF=0 and ZF=0).
REX + OF 97	SETA <i>r/m8</i> *	M	Valid	N.E.	Set byte if above (CF=0 and ZF=0).
OF 93	SETAE <i>r/m8</i>	M	Valid	Valid	Set byte if above or equal (CF=0).
REX + OF 93	SETAE <i>r/m8</i> *	M	Valid	N.E.	Set byte if above or equal (CF=0).
OF 92	SETB <i>r/m8</i>	M	Valid	Valid	Set byte if below (CF=1).
REX + OF 92	SETB <i>r/m8</i> *	M	Valid	N.E.	Set byte if below (CF=1).
OF 96	SETBE <i>r/m8</i>	M	Valid	Valid	Set byte if below or equal (CF=1 or ZF=1).
REX + OF 96	SETBE <i>r/m8</i> *	M	Valid	N.E.	Set byte if below or equal (CF=1 or ZF=1).
OF 92	SETC <i>r/m8</i>	M	Valid	Valid	Set byte if carry (CF=1).
REX + OF 92	SETC <i>r/m8</i> *	M	Valid	N.E.	Set byte if carry (CF=1).
OF 94	SETE <i>r/m8</i>	M	Valid	Valid	Set byte if equal (ZF=1).
REX + OF 94	SETE <i>r/m8</i> *	M	Valid	N.E.	Set byte if equal (ZF=1).
OF 9F	SETG <i>r/m8</i>	M	Valid	Valid	Set byte if greater (ZF=0 and SF=0F).
REX + OF 9F	SETG <i>r/m8</i> *	M	Valid	N.E.	Set byte if greater (ZF=0 and SF=0F).
OF 9D	SETGE <i>r/m8</i>	M	Valid	Valid	Set byte if greater or equal (SF=0F).
REX + OF 9D	SETGE <i>r/m8</i> *	M	Valid	N.E.	Set byte if greater or equal (SF=0F).
OF 9C	SETL <i>r/m8</i>	M	Valid	Valid	Set byte if less (SF≠ 0F).
REX + OF 9C	SETL <i>r/m8</i> *	M	Valid	N.E.	Set byte if less (SF≠ 0F).
OF 9E	SETLE <i>r/m8</i>	M	Valid	Valid	Set byte if less or equal (ZF=1 or SF≠ 0F).
REX + OF 9E	SETLE <i>r/m8</i> *	M	Valid	N.E.	Set byte if less or equal (ZF=1 or SF≠ 0F).
OF 96	SETNA <i>r/m8</i>	M	Valid	Valid	Set byte if not above (CF=1 or ZF=1).
REX + OF 96	SETNA <i>r/m8</i> *	M	Valid	N.E.	Set byte if not above (CF=1 or ZF=1).
OF 92	SETNAE <i>r/m8</i>	M	Valid	Valid	Set byte if not above or equal (CF=1).
REX + OF 92	SETNAE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not above or equal (CF=1).
OF 93	SETNB <i>r/m8</i>	M	Valid	Valid	Set byte if not below (CF=0).
REX + OF 93	SETNB <i>r/m8</i> *	M	Valid	N.E.	Set byte if not below (CF=0).
OF 97	SETNBE <i>r/m8</i>	M	Valid	Valid	Set byte if not below or equal (CF=0 and ZF=0).
REX + OF 97	SETNBE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not below or equal (CF=0 and ZF=0).
OF 93	SETNC <i>r/m8</i>	M	Valid	Valid	Set byte if not carry (CF=0).
REX + OF 93	SETNC <i>r/m8</i> *	M	Valid	N.E.	Set byte if not carry (CF=0).
OF 95	SETNE <i>r/m8</i>	M	Valid	Valid	Set byte if not equal (ZF=0).
REX + OF 95	SETNE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not equal (ZF=0).
OF 9E	SETNG <i>r/m8</i>	M	Valid	Valid	Set byte if not greater (ZF=1 or SF≠ 0F)
REX + OF 9E	SETNG <i>r/m8</i> *	M	Valid	N.E.	Set byte if not greater (ZF=1 or SF≠ 0F).
OF 9C	SETNGE <i>r/m8</i>	M	Valid	Valid	Set byte if not greater or equal (SF≠ 0F).
REX + OF 9C	SETNGE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not greater or equal (SF≠ 0F).
OF 9D	SETNL <i>r/m8</i>	M	Valid	Valid	Set byte if not less (SF=0F).
REX + OF 9D	SETNL <i>r/m8</i> *	M	Valid	N.E.	Set byte if not less (SF=0F).
OF 9F	SETNLE <i>r/m8</i>	M	Valid	Valid	Set byte if not less or equal (ZF=0 and SF=0F).

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
REX + 0F 9F	SETNLE <i>r/m8</i> *	M	Valid	N.E.	Set byte if not less or equal (ZF=0 and SF=OF).
0F 91	SETNO <i>r/m8</i>	M	Valid	Valid	Set byte if not overflow (OF=0).
REX + 0F 91	SETNO <i>r/m8</i> *	M	Valid	N.E.	Set byte if not overflow (OF=0).
0F 9B	SETNP <i>r/m8</i>	M	Valid	Valid	Set byte if not parity (PF=0).
REX + 0F 9B	SETNP <i>r/m8</i> *	M	Valid	N.E.	Set byte if not parity (PF=0).
0F 99	SETNS <i>r/m8</i>	M	Valid	Valid	Set byte if not sign (SF=0).
REX + 0F 99	SETNS <i>r/m8</i> *	M	Valid	N.E.	Set byte if not sign (SF=0).
0F 95	SETNZ <i>r/m8</i>	M	Valid	Valid	Set byte if not zero (ZF=0).
REX + 0F 95	SETNZ <i>r/m8</i> *	M	Valid	N.E.	Set byte if not zero (ZF=0).
0F 90	SETO <i>r/m8</i>	M	Valid	Valid	Set byte if overflow (OF=1)
REX + 0F 90	SETO <i>r/m8</i> *	M	Valid	N.E.	Set byte if overflow (OF=1).
0F 9A	SETP <i>r/m8</i>	M	Valid	Valid	Set byte if parity (PF=1).
REX + 0F 9A	SETP <i>r/m8</i> *	M	Valid	N.E.	Set byte if parity (PF=1).
0F 9A	SETPE <i>r/m8</i>	M	Valid	Valid	Set byte if parity even (PF=1).
REX + 0F 9A	SETPE <i>r/m8</i> *	M	Valid	N.E.	Set byte if parity even (PF=1).
0F 9B	SETPO <i>r/m8</i>	M	Valid	Valid	Set byte if parity odd (PF=0).
REX + 0F 9B	SETPO <i>r/m8</i> *	M	Valid	N.E.	Set byte if parity odd (PF=0).
0F 98	SETS <i>r/m8</i>	M	Valid	Valid	Set byte if sign (SF=1).
REX + 0F 98	SETS <i>r/m8</i> *	M	Valid	N.E.	Set byte if sign (SF=1).
0F 94	SETZ <i>r/m8</i>	M	Valid	Valid	Set byte if zero (ZF=1).
REX + 0F 94	SETZ <i>r/m8</i> *	M	Valid	N.E.	Set byte if zero (ZF=1).
<b>NOTES:</b>					
* In 64-bit mode, <i>r/m8</i> can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.					

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms “above” and “below” are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms “greater” and “less” are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SET<sub>cc</sub> instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, “EFLAGS Condition Codes,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SET<sub>cc</sub> instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

The reg field of the ModR/M byte is not used for the SETCC instruction and those opcode bits are ignored by the processor.

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction's operation is the same as in legacy mode and compatibility mode.

## Operation

IF condition  
     THEN DEST  $\leftarrow$  1;  
     ELSE DEST  $\leftarrow$  0;  
 FI;

## Flags Affected

None.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

## SFENCE—Store Fence

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP 0F AE F8	SFENCE	Z0	Valid	Valid	Serializes store operations.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Orders processor execution relative to all memory stores prior the SFENCE instruction. The processor ensures that every store prior to SFENCE is globally visible before any store after SFENCE becomes globally visible. The SFENCE instruction is ordered with respect to memory stores, other SFENCE instructions, MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to memory loads or the LFENCE instruction.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, SFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 8-F.

### Operation

Wait\_On\_Following\_Stores\_Until(preceding\_stores\_globally\_visible);

### Intel C/C++ Compiler Intrinsic Equivalent

void \_mm\_sfence(void)

### Exceptions (All Operating Modes)

#UD                      If CPUID.01H:EDX.SSE[bit 25] = 0.  
                              If the LOCK prefix is used.

## SGDT—Store Global Descriptor Table Register

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 /0	SGDT <i>m</i>	M	Valid	Valid	Store GDTR to <i>m</i> .

### NOTES:

\* See IA-32 Architecture Compatibility section below.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 or 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

### IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

### Operation

IF instruction is SGDT

IF OperandSize = 16 or OperandSize = 32 (\* Legacy or Compatibility Mode \*)

THEN

DEST[0:15] ← GDTR(Limit);

DEST[16:47] ← GDTR(Base); (\* Full 32-bit base address stored \*)

FI;

ELSE (\* 64-bit Mode \*)

DEST[0:15] ← GDTR(Limit);

DEST[16:79] ← GDTR(Base); (\* Full 64-bit base address stored \*)

FI;

FI;

### Flags Affected

None.

**Protected Mode Exceptions**

#UD	If the LOCK prefix is used.
#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

**Real-Address Mode Exceptions**

#UD	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

**Virtual-8086 Mode Exceptions**

#UD	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form. If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

## SHA1RND4—Perform Four Rounds of SHA1 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 3A CC /r ib SHA1RND4 xmm1, xmm2/m128, imm8	RMI	V/V	SHA	Performs four rounds of SHA1 operation operating on SHA1 state (A,B,C,D) from xmm1, with a pre-computed sum of the next 4 round message dwords and state variable E from xmm2/m128. The immediate byte controls logic functions and round constants.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8

### Description

The SHA1RND4 instruction performs four rounds of SHA1 operation using an initial SHA1 state (A,B,C,D) from the first operand (which is a source operand and the destination operand) and some pre-computed sum of the next 4 round message dwords, and state variable E from the second operand (a source operand). The updated SHA1 state (A,B,C,D) after four rounds of processing is stored in the destination operand.

### Operation

#### SHA1RND4

The function  $f()$  and Constant  $K$  are dependent on the value of the immediate.

```
IF ( imm8[1:0] = 0 )
    THEN  $f() \leftarrow f_0()$ ,  $K \leftarrow K_0$ ;
ELSE IF ( imm8[1:0] = 1 )
    THEN  $f() \leftarrow f_1()$ ,  $K \leftarrow K_1$ ;
ELSE IF ( imm8[1:0] = 2 )
    THEN  $f() \leftarrow f_2()$ ,  $K \leftarrow K_2$ ;
ELSE IF ( imm8[1:0] = 3 )
    THEN  $f() \leftarrow f_3()$ ,  $K \leftarrow K_3$ ;
FI;
```

```
A  $\leftarrow$  SRC1[127:96];
B  $\leftarrow$  SRC1[95:64];
C  $\leftarrow$  SRC1[63:32];
D  $\leftarrow$  SRC1[31:0];
W0E  $\leftarrow$  SRC2[127:96];
W1  $\leftarrow$  SRC2[95:64];
W2  $\leftarrow$  SRC2[63:32];
W3  $\leftarrow$  SRC2[31:0];
```

Round  $i = 0$  operation:

```
A_1  $\leftarrow$   $f(B, C, D) + (A \text{ ROL } 5) + W_0E + K$ ;
B_1  $\leftarrow$  A;
C_1  $\leftarrow$  B ROL 30;
D_1  $\leftarrow$  C;
E_1  $\leftarrow$  D;
```

FOR  $i = 1$  to 3

```
A_(i+1)  $\leftarrow$   $f(B_i, C_i, D_i) + (A_i \text{ ROL } 5) + W_i + E_i + K$ ;
B_(i+1)  $\leftarrow$  Ai;
```

```

    C_(i + 1) ← B_i ROL 30;
    D_(i + 1) ← C_i;
    E_(i + 1) ← D_i;
ENDFOR

```

```

DEST[127:96] ← A_4;
DEST[95:64] ← B_4;
DEST[63:32] ← C_4;
DEST[31:0] ← D_4;

```

#### Intel C/C++ Compiler Intrinsic Equivalent

SHA1RND4: `__m128i _mm_sha1rnds4_epu32(__m128i, __m128i, const int);`

#### Flags Affected

None

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

See Exceptions Type 4.



## SHA1NEXTE—Calculate SHA1 State Variable E after Four Rounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 38 C8 /r SHA1NEXTE xmm1, xmm2/m128	RM	V/V	SHA	Calculates SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in xmm1. The calculated value of the SHA1 state variable E is added to the scheduled dwords in xmm2/m128, and stored with some of the scheduled dwords in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

### Description

The SHA1NEXTE calculates the SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in the destination operand. The calculated value of the SHA1 state variable E is added to the source operand, which contains the scheduled dwords.

### Operation

#### SHA1NEXTE

$TMP \leftarrow (SRC1[127:96] \text{ ROL } 30);$

$DEST[127:96] \leftarrow SRC2[127:96] + TMP;$

$DEST[95:64] \leftarrow SRC2[95:64];$

$DEST[63:32] \leftarrow SRC2[63:32];$

$DEST[31:0] \leftarrow SRC2[31:0];$

### Intel C/C++ Compiler Intrinsic Equivalent

SHA1NEXTE: `__m128i __mm_sha1nexte_epu32(__m128i, __m128i);`

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

## SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 C9 /r SHA1MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA1 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

### Description

The SHA1MSG1 instruction is one of two SHA1 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA1 message dwords.

### Operation

#### SHA1MSG1

```

W0 ← SRC1[127:96];
W1 ← SRC1[95:64];
W2 ← SRC1[63:32];
W3 ← SRC1[31:0];
W4 ← SRC2[127:96];
W5 ← SRC2[95:64];

```

```

DEST[127:96] ← W2 XOR W0;
DEST[95:64] ← W3 XOR W1;
DEST[63:32] ← W4 XOR W2;
DEST[31:0] ← W5 XOR W3;

```

### Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG1: \_\_m128i \_mm\_sha1msg1\_epu32(\_\_m128i, \_\_m128i);

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

## SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 38 CA /r SHA1MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA1 message dwords using intermediate results from xmm1 and the previous message dwords from xmm2/m128, storing the result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

### Description

The SHA1MSG2 instruction is one of two SHA1 message scheduling instructions. The instruction performs the final calculation to derive the next four SHA1 message dwords.

### Operation

#### SHA1MSG2

```

W13 ← SRC2[95:64];
W14 ← SRC2[63:32];
W15 ← SRC2[31:0];
W16 ← (SRC1[127:96] XOR W13) ROL 1;
W17 ← (SRC1[95:64] XOR W14) ROL 1;
W18 ← (SRC1[63:32] XOR W15) ROL 1;
W19 ← (SRC1[31:0] XOR W16) ROL 1;

```

```

DEST[127:96] ← W16;
DEST[95:64] ← W17;
DEST[63:32] ← W18;
DEST[31:0] ← W19;

```

### Intel C/C++ Compiler Intrinsic Equivalent

```
SHA1MSG2: __m128i _mm_sha1msg2_epu32(__m128i, __m128i);
```

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

## SHA256RND2—Perform Two Rounds of SHA256 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CB /r SHA256RND2 xmm1, xmm2/m128, <XMM0>	RM0	V/V	SHA	Perform 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from xmm1, an initial SHA256 state (A,B,E,F) from xmm2/m128, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Implicit XMM0 (r)

### Description

The SHA256RND2 instruction performs 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from the first operand, an initial SHA256 state (A,B,E,F) from the second operand, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand xmm0. Note that only the two lower dwords of XMM0 are used by the instruction.

The updated SHA256 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

### Operation

#### SHA256RND2

```

A_0 ← SRC2[127:96];
B_0 ← SRC2[95:64];
C_0 ← SRC1[127:96];
D_0 ← SRC1[95:64];
E_0 ← SRC2[63:32];
F_0 ← SRC2[31:0];
G_0 ← SRC1[63:32];
H_0 ← SRC1[31:0];
WK_0 ← XMM0[31:0];
WK_1 ← XMM0[63:32];

```

FOR i = 0 to 1

```

  A_(i+1) ← Ch(E_i, F_i, G_i) + Σ1( E_i ) + WKi + H_i + Maj(A_i, B_i, C_i) + Σ0( A_i );
  B_(i+1) ← A_i;
  C_(i+1) ← B_i;
  D_(i+1) ← C_i;
  E_(i+1) ← Ch(E_i, F_i, G_i) + Σ1( E_i ) + WKi + H_i + D_i;
  F_(i+1) ← E_i;
  G_(i+1) ← F_i;
  H_(i+1) ← G_i;

```

ENDFOR

```

DEST[127:96] ← A_2;
DEST[95:64] ← B_2;
DEST[63:32] ← E_2;
DEST[31:0] ← F_2;

```

### Intel C/C++ Compiler Intrinsic Equivalent

SHA256RNDs2: `__m128i _mm_sha256rnds2_epu32(__m128i, __m128i, __m128i);`

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

## SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 38 CC /r SHA256MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

### Description

The SHA256MSG1 instruction is one of two SHA256 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA256 message dwords.

### Operation

#### SHA256MSG1

$$\begin{aligned} W4 &\leftarrow \text{SRC2}[31:0]; \\ W3 &\leftarrow \text{SRC1}[127:96]; \\ W2 &\leftarrow \text{SRC1}[95:64]; \\ W1 &\leftarrow \text{SRC1}[63:32]; \\ W0 &\leftarrow \text{SRC1}[31:0]; \end{aligned}$$

$$\begin{aligned} \text{DEST}[127:96] &\leftarrow W3 + \sigma_0(W4); \\ \text{DEST}[95:64] &\leftarrow W2 + \sigma_0(W3); \\ \text{DEST}[63:32] &\leftarrow W1 + \sigma_0(W2); \\ \text{DEST}[31:0] &\leftarrow W0 + \sigma_0(W1); \end{aligned}$$

### Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG1: \_\_m128i\_mm\_sha256msg1\_epu32(\_\_m128i, \_\_m128i);

### Flags Affected

None

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 4.

**SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords**

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CD /r SHA256MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

**Description**

The SHA256MSG2 instruction is one of two SHA2 message scheduling instructions. The instruction performs the final calculation for the next four SHA256 message dwords.

**Operation****SHA256MSG2**

$$\begin{aligned} W14 &\leftarrow \text{SRC2}[95:64]; \\ W15 &\leftarrow \text{SRC2}[127:96]; \\ W16 &\leftarrow \text{SRC1}[31:0] + \sigma_1(W14); \\ W17 &\leftarrow \text{SRC1}[63:32] + \sigma_1(W15); \\ W18 &\leftarrow \text{SRC1}[95:64] + \sigma_1(W16); \\ W19 &\leftarrow \text{SRC1}[127:96] + \sigma_1(W17); \end{aligned}$$

$$\begin{aligned} \text{DEST}[127:96] &\leftarrow W19; \\ \text{DEST}[95:64] &\leftarrow W18; \\ \text{DEST}[63:32] &\leftarrow W17; \\ \text{DEST}[31:0] &\leftarrow W16; \end{aligned}$$
**Intel C/C++ Compiler Intrinsic Equivalent**

SHA256MSG2: \_\_m128i \_mm\_sha256msg2\_epu32(\_\_m128i, \_\_m128i);

**Flags Affected**

None

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

## SHLD—Double Precision Shift Left

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF A4 /r ib	SHLD <i>r/m16, r16, imm8</i>	MRI	Valid	Valid	Shift <i>r/m16</i> to left <i>imm8</i> places while shifting bits from <i>r16</i> in from the right.
OF A5 /r	SHLD <i>r/m16, r16, CL</i>	MRC	Valid	Valid	Shift <i>r/m16</i> to left CL places while shifting bits from <i>r16</i> in from the right.
OF A4 /r ib	SHLD <i>r/m32, r32, imm8</i>	MRI	Valid	Valid	Shift <i>r/m32</i> to left <i>imm8</i> places while shifting bits from <i>r32</i> in from the right.
REX.W + OF A4 /r ib	SHLD <i>r/m64, r64, imm8</i>	MRI	Valid	N.E.	Shift <i>r/m64</i> to left <i>imm8</i> places while shifting bits from <i>r64</i> in from the right.
OF A5 /r	SHLD <i>r/m32, r32, CL</i>	MRC	Valid	Valid	Shift <i>r/m32</i> to left CL places while shifting bits from <i>r32</i> in from the right.
REX.W + OF A5 /r	SHLD <i>r/m64, r64, CL</i>	MRC	Valid	N.E.	Shift <i>r/m64</i> to left CL places while shifting bits from <i>r64</i> in from the right.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

### Description

The SHLD instruction is used for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or in the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode; only bits 0 through 4 of the count are used. This masks the count to a value between 0 and 31. If a count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```

IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE

```



```

IF COUNT > SIZE
    THEN (* Bad parameters *)
        DEST is undefined;
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE (* Perform the shift *)
        CF ← BIT[DEST, SIZE - COUNT];
        (* Last bit shifted out on exit *)
        FOR i ← SIZE - 1 DOWN TO COUNT
            DO
                Bit(DEST, i) ← Bit(DEST, i - COUNT);
            OD;
        FOR i ← COUNT - 1 DOWN TO 0
            DO
                BIT[DEST, i] ← BIT[DEST, i - COUNT + SIZE];
            OD;
        FI;
FI;

```

### Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## SHRD—Double Precision Shift Right

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF AC /r ib	SHRD <i>r/m16, r16, imm8</i>	MRI	Valid	Valid	Shift <i>r/m16</i> to right <i>imm8</i> places while shifting bits from <i>r16</i> in from the left.
OF AD /r	SHRD <i>r/m16, r16, CL</i>	MRC	Valid	Valid	Shift <i>r/m16</i> to right CL places while shifting bits from <i>r16</i> in from the left.
OF AC /r ib	SHRD <i>r/m32, r32, imm8</i>	MRI	Valid	Valid	Shift <i>r/m32</i> to right <i>imm8</i> places while shifting bits from <i>r32</i> in from the left.
REX.W + OF AC /r ib	SHRD <i>r/m64, r64, imm8</i>	MRI	Valid	N.E.	Shift <i>r/m64</i> to right <i>imm8</i> places while shifting bits from <i>r64</i> in from the left.
OF AD /r	SHRD <i>r/m32, r32, CL</i>	MRC	Valid	Valid	Shift <i>r/m32</i> to right CL places while shifting bits from <i>r32</i> in from the left.
REX.W + OF AD /r	SHRD <i>r/m64, r64, CL</i>	MRC	Valid	N.E.	Shift <i>r/m64</i> to right CL places while shifting bits from <i>r64</i> in from the left.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

## Description

The SHRD instruction is useful for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode, the width of the count mask is 5 bits. Only bits 0 through 4 of the count register are used (masking the count to a value between 0 and 31). If the count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

## Operation

```

IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE

```

```

IF COUNT > SIZE
    THEN (* Bad parameters *)
        DEST is undefined;
        CF, OF, SF, ZF, AF, PF are undefined;
    ELSE (* Perform the shift *)
        CF ← BIT[DEST, COUNT - 1]; (* Last bit shifted out on exit *)
        FOR i ← 0 TO SIZE - 1 - COUNT
            DO
                BIT[DEST, i] ← BIT[DEST, i + COUNT];
            OD;
        FOR i ← SIZE - COUNT TO SIZE - 1
            DO
                BIT[DEST, i] ← BIT[SRC, i + COUNT - SIZE];
            OD;
        FI;
    FI;
FI;

```

### Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## SHUFPD—Packed Interleave Shuffle of Pairs of Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /r ib SHUFPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle two pairs of double-precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.NDS.128.66.0F.WIG C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Shuffle two pairs of double-precision floating-point values from xmm2 and xmm3/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.NDS.256.66.0F.WIG C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Shuffle four pairs of double-precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in ymm1.
EVEX.NDS.128.66.0F.W1 C6 /r ib VSHUFPD xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Shuffle two pairs of double-precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8 to select from each pair. store interleaved results in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 C6 /r ib VSHUFPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Shuffle four pairs of double-precision floating-point values from ymm2 and ymm3/m256/m64bcst using imm8 to select from each pair. store interleaved results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 C6 /r ib VSHUFPD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F	Shuffle eight pairs of double-precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8 to select from each pair. store interleaved results in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	Imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Selects a double-precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double-precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to receive the shuffled result of an input pair.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The select controls are the bit 3:0 of the imm8 byte, imm8[7:4) are ignored.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

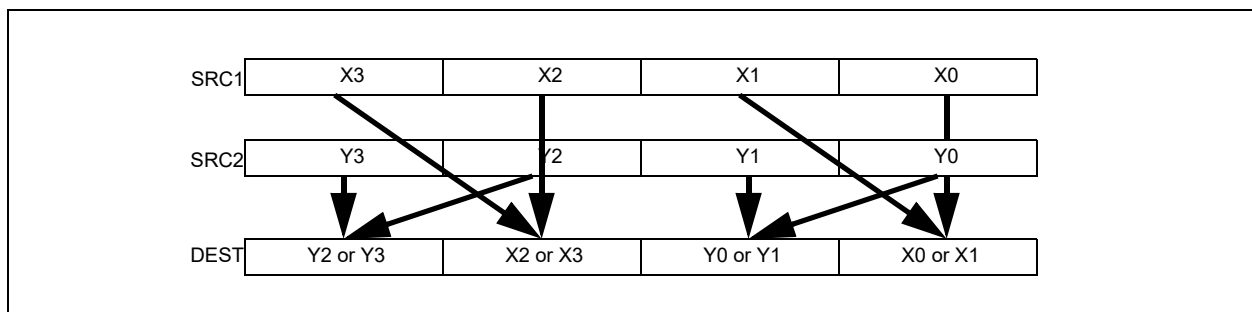


Figure 4-25. 256-bit VSHUFPD Operation of Four Pairs of DP FP Values

### Operation

#### VSHUFPD (EVEX encoded versions when SRC2 is a vector register)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF IMMO[0] = 0

THEN TMP\_DEST[63:0] ← SRC1[63:0]

ELSE TMP\_DEST[63:0] ← SRC1[127:64] FI;

IF IMMO[1] = 0

THEN TMP\_DEST[127:64] ← SRC2[63:0]

ELSE TMP\_DEST[127:64] ← SRC2[127:64] FI;

IF VL ≥ 256

IF IMMO[2] = 0

THEN TMP\_DEST[191:128] ← SRC1[191:128]

ELSE TMP\_DEST[191:128] ← SRC1[255:192] FI;

IF IMMO[3] = 0

THEN TMP\_DEST[255:192] ← SRC2[191:128]

ELSE TMP\_DEST[255:192] ← SRC2[255:192] FI;

FI;

IF VL ≥ 512

IF IMMO[4] = 0

THEN TMP\_DEST[319:256] ← SRC1[319:256]

ELSE TMP\_DEST[319:256] ← SRC1[383:320] FI;

IF IMMO[5] = 0

THEN TMP\_DEST[383:320] ← SRC2[319:256]

ELSE TMP\_DEST[383:320] ← SRC2[383:320] FI;

IF IMMO[6] = 0

THEN TMP\_DEST[447:384] ← SRC1[447:384]

ELSE TMP\_DEST[447:384] ← SRC1[511:448] FI;

IF IMMO[7] = 0

THEN TMP\_DEST[511:448] ← SRC2[447:384]

ELSE TMP\_DEST[511:448] ← SRC2[511:448] FI;

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

```

        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

#### VSHUFPD (EVEX encoded versions when SRC2 is memory)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF IMMO[0] = 0
    THEN TMP_DEST[63:0] ← SRC1[63:0]
    ELSE TMP_DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
    THEN TMP_DEST[127:64] ← TMP_SRC2[63:0]
    ELSE TMP_DEST[127:64] ← TMP_SRC2[127:64] FI;
IF VL >= 256
    IF IMMO[2] = 0
        THEN TMP_DEST[191:128] ← SRC1[191:128]
        ELSE TMP_DEST[191:128] ← SRC1[255:192] FI;
    IF IMMO[3] = 0
        THEN TMP_DEST[255:192] ← TMP_SRC2[191:128]
        ELSE TMP_DEST[255:192] ← TMP_SRC2[255:192] FI;
    FI;
IF VL >= 512
    IF IMMO[4] = 0
        THEN TMP_DEST[319:256] ← SRC1[319:256]
        ELSE TMP_DEST[319:256] ← SRC1[383:320] FI;
    IF IMMO[5] = 0
        THEN TMP_DEST[383:320] ← TMP_SRC2[319:256]
        ELSE TMP_DEST[383:320] ← TMP_SRC2[383:320] FI;
    IF IMMO[6] = 0
        THEN TMP_DEST[447:384] ← SRC1[447:384]
        ELSE TMP_DEST[447:384] ← SRC1[511:448] FI;
    IF IMMO[7] = 0
        THEN TMP_DEST[511:448] ← TMP_SRC2[447:384]
        ELSE TMP_DEST[511:448] ← TMP_SRC2[511:448] FI;
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] ← 0
        FI
    FI

```



```

        ELSE *zeroing-masking*          ; zeroing-masking
          DEST[i+63:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VSHUFPD (VEX.256 encoded version)**

```

IF IMMO[0] = 0
  THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
  THEN DEST[127:64] ← SRC2[63:0]
  ELSE DEST[127:64] ← SRC2[127:64] FI;
IF IMMO[2] = 0
  THEN DEST[191:128] ← SRC1[191:128]
  ELSE DEST[191:128] ← SRC1[255:192] FI;
IF IMMO[3] = 0
  THEN DEST[255:192] ← SRC2[191:128]
  ELSE DEST[255:192] ← SRC2[255:192] FI;
DEST[MAXVL-1:256] (Unmodified)

```

**VSHUFPD (VEX.128 encoded version)**

```

IF IMMO[0] = 0
  THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
  THEN DEST[127:64] ← SRC2[63:0]
  ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[MAXVL-1:128] ← 0

```

**VSHUFPD (128-bit Legacy SSE version)**

```

IF IMMO[0] = 0
  THEN DEST[63:0] ← SRC1[63:0]
  ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMMO[1] = 0
  THEN DEST[127:64] ← SRC2[63:0]
  ELSE DEST[127:64] ← SRC2[127:64] FI;
DEST[MAXVL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSHUFPD __m512d _mm512_shuffle_pd(__m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_mask_shuffle_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_maskz_shuffle_pd(__mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m256d _mm256_shuffle_pd(__m256d a, __m256d b, const int select);
VSHUFPD __m256d _mm256_mask_shuffle_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VSHUFPD __m256d _mm256_maskz_shuffle_pd(__mmask8 k, __m256d a, __m256d b, int imm);
SHUFPD __m128d _mm_shuffle_pd(__m128d a, __m128d b, const int select);
VSHUFPD __m128d _mm_mask_shuffle_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VSHUFPD __m128d _mm_maskz_shuffle_pd(__mmask8 k, __m128d a, __m128d b, int imm);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

## SHUFPS—Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	A	V/V	SSE	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.NDS.128.OF.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.NDS.256.OF.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.
EVEX.NDS.128.OF.W0 C6 /r ib VSHUFPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.
EVEX.NDS.256.OF.W0 C6 /r ib VSHUFPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.
EVEX.NDS.512.OF.W0 C6 /r ib VSHUFPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	C	V/V	AVX512F	Select from quadruplet of single-precision floating-point values in zmm2 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	Imm8
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

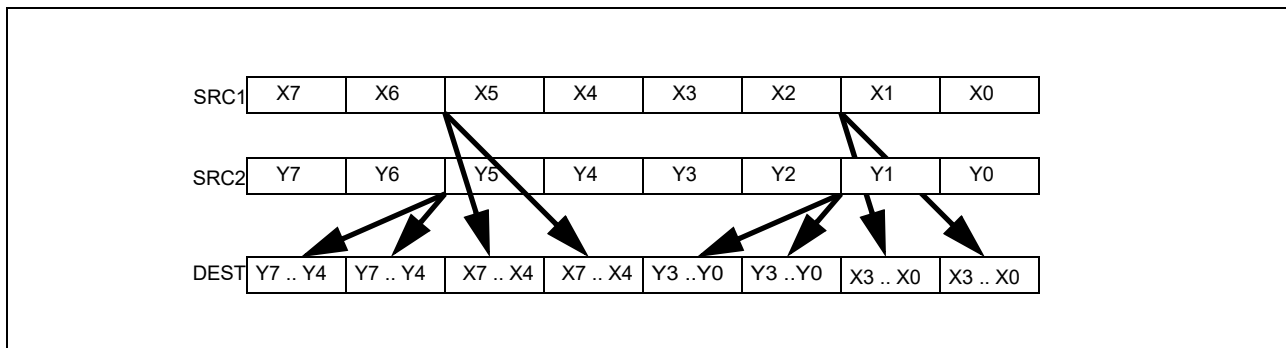
Selects a single-precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. Imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for the high and low 128-bit of the destination.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.



**Figure 4-26. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result**

### Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
  0: TMP ← SRC[31:0];
  1: TMP ← SRC[63:32];
  2: TMP ← SRC[95:64];
  3: TMP ← SRC[127:96];
ESAC;
RETURN TMP
}
```

### VPSHUFPS (EVEX encoded versions when SRC2 is a vector register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
  TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
  TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
  TMP_DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
  TMP_DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
  TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
  TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
  TMP_DEST[351:320] ← Select4(SRC2[383:256], imm8[5:4]);
  TMP_DEST[383:352] ← Select4(SRC2[383:256], imm8[7:6]);
  TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
  TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
  TMP_DEST[479:448] ← Select4(SRC2[511:384], imm8[5:4]);
  TMP_DEST[511:480] ← Select4(SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
```

```

    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPSHUFPS (EVEX encoded versions when SRC2 is memory)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

```

    i ← j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL ≥ 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] ← Select4(TMP_SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] ← Select4(TMP_SRC2[255:128], imm8[7:6]);
FI;
IF VL ≥ 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] ← Select4(TMP_SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] ← Select4(TMP_SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] ← Select4(TMP_SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] ← Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;

```

```

FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR

```

DEST[MAXVL-1:VL]  $\leftarrow$  0

#### VSHUFPS (VEX.256 encoded version)

DEST[31:0]  $\leftarrow$  Select4(SRC1[127:0], imm8[1:0]);  
 DEST[63:32]  $\leftarrow$  Select4(SRC1[127:0], imm8[3:2]);  
 DEST[95:64]  $\leftarrow$  Select4(SRC2[127:0], imm8[5:4]);  
 DEST[127:96]  $\leftarrow$  Select4(SRC2[127:0], imm8[7:6]);  
 DEST[159:128]  $\leftarrow$  Select4(SRC1[255:128], imm8[1:0]);  
 DEST[191:160]  $\leftarrow$  Select4(SRC1[255:128], imm8[3:2]);  
 DEST[223:192]  $\leftarrow$  Select4(SRC2[255:128], imm8[5:4]);  
 DEST[255:224]  $\leftarrow$  Select4(SRC2[255:128], imm8[7:6]);  
 DEST[MAXVL-1:256]  $\leftarrow$  0

#### VSHUFPS (VEX.128 encoded version)

DEST[31:0]  $\leftarrow$  Select4(SRC1[127:0], imm8[1:0]);  
 DEST[63:32]  $\leftarrow$  Select4(SRC1[127:0], imm8[3:2]);  
 DEST[95:64]  $\leftarrow$  Select4(SRC2[127:0], imm8[5:4]);  
 DEST[127:96]  $\leftarrow$  Select4(SRC2[127:0], imm8[7:6]);  
 DEST[MAXVL-1:128]  $\leftarrow$  0

#### SHUFPS (128-bit Legacy SSE version)

DEST[31:0]  $\leftarrow$  Select4(SRC1[127:0], imm8[1:0]);  
 DEST[63:32]  $\leftarrow$  Select4(SRC1[127:0], imm8[3:2]);  
 DEST[95:64]  $\leftarrow$  Select4(SRC2[127:0], imm8[5:4]);  
 DEST[127:96]  $\leftarrow$  Select4(SRC2[127:0], imm8[7:6]);  
 DEST[MAXVL-1:128] (Unmodified)

#### Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPS \_\_m512 \_mm512\_shuffle\_ps(\_\_m512 a, \_\_m512 b, int imm);  
 VSHUFPS \_\_m512 \_mm512\_mask\_shuffle\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b, int imm);  
 VSHUFPS \_\_m512 \_mm512\_maskz\_shuffle\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b, int imm);  
 VSHUFPS \_\_m256 \_mm256\_shuffle\_ps (\_\_m256 a, \_\_m256 b, const int select);  
 VSHUFPS \_\_m256 \_mm256\_mask\_shuffle\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b, int imm);  
 VSHUFPS \_\_m256 \_mm256\_maskz\_shuffle\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256 b, int imm);  
 SHUFPS \_\_m128 \_mm\_shuffle\_ps (\_\_m128 a, \_\_m128 b, const int select);  
 VSHUFPS \_\_m128 \_mm\_mask\_shuffle\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b, int imm);  
 VSHUFPS \_\_m128 \_mm\_maskz\_shuffle\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128 b, int imm);

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

SIDT—Store Interrupt Descriptor Table Register

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 /1	SIDT <i>m</i>	M	Valid	Valid	Store IDTR to <i>m</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m ( <i>w</i> )	NA	NA	NA

Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See “LGDT/LIDT—Load Global/Interrupt Descriptor Table Register” in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

Operation

```
IF instruction is SIDT
  THEN
    IF OperandSize =16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
      THEN
        DEST[0:15] ← IDTR(Limit);
        DEST[16:47] ← IDTR(Base); FI; (* Full 32-bit base address stored *)
      ELSE (* 64-bit Mode *)
        DEST[0:15] ← IDTR(Limit);
        DEST[16:79] ← IDTR(Base); (* Full 64-bit base address stored *)
      FI;
    FI;
```

Flags Affected

None.

**Protected Mode Exceptions**

#GP(0)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If CR4.UMIP = 1 and CPL &gt; 0.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If CR4.UMIP = 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the LOCK prefix is used.
#GP(0)	<p>If the memory address is in a non-canonical form.</p> <p>If CR4.UMIP = 1 and CPL &gt; 0.</p>
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.



## SLDT—Store Local Descriptor Table Register

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /0	SLDT <i>r/m16</i>	M	Valid	Valid	Stores segment selector from LDTR in <i>r/m16</i> .
REX.W + OF 00 /0	SLDT <i>r64/m16</i>	M	Valid	Valid	Stores segment selector from LDTR in <i>r64/m16</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

Outside IA-32e mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors. They are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In compatibility mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). The behavior of SLDT with a 64-bit register is to zero-extend the 16-bit selector and store it in the register. If the destination is memory and operand size is 64, SLDT will write the 16-bit selector to memory as a 16-bit quantity, regardless of the operand size.

### Operation

DEST ← LDTR(SegmentSelector);

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If CR4.UIMP = 1 and CPL &gt; 0.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD                      The SLDT instruction is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD                      The SLDT instruction is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

## SMSW—Store Machine Status Word

Opcode*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 /4	SMSW <i>r/m16</i>	M	Valid	Valid	Store machine status word to <i>r/m16</i> .
OF 01 /4	SMSW <i>r32/m16</i>	M	Valid	Valid	Store machine status word in low-order 16 bits of <i>r32/m16</i> ; high-order 16 bits of <i>r32</i> are undefined.
REX.W + OF 01 /4	SMSW <i>r64/m16</i>	M	Valid	Valid	Store machine status word in low-order 16 bits of <i>r64/m16</i> ; high-order 16 bits of <i>r32</i> are undefined.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a general-purpose register or a memory location.

In non-64-bit modes, when the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the high-order 16 bits are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, the behavior of the SMSW instruction is defined by the following examples:

- SMSW *r16* operand size 16, store CR0[15:0] in *r16*
- SMSW *r32* operand size 32, zero-extend CR0[31:0], and store in *r32*
- SMSW *r64* operand size 64, zero-extend CR0[63:0], and store in *r64*
- SMSW *m16* operand size 16, store CR0[15:0] in *m16*
- SMSW *m16* operand size 32, store CR0[15:0] in *m16* (not *m32*)
- SMSW *m16* operands size 64, store CR0[15:0] in *m16* (not *m64*)

SMSW is only useful in operating-system software. However, it is not a privileged instruction and can be used in application programs if CR4.UMIP = 0. It is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on IA-32 and Intel 64 processors beginning with the Intel386 processors should use the MOV CR instruction to load the machine status word.

See “Changes to Instruction Behavior in VMX Non-Root Operation” in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

### Operation

DEST ← CR0[15:0];

(\* Machine status word \*)

### Flags Affected

None.

**Protected Mode Exceptions**

#GP(0)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.</p> <p>If CR4.UMIP = 1 and CPL &gt; 0.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If CR4.UMIP = 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	<p>If the memory address is in a non-canonical form.</p> <p>If CR4.UMIP = 1 and CPL &gt; 0.</p>
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

## SQRTPD—Square Root of Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 51 /r SQRTPD xmm1, xmm2/m128	A	V/V	SSE2	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F.WIG 51 /r VSQRTPD xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.66.0F.WIG 51 /r VSQRTPD ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.66.0F.W1 51 /r VSQRTPD xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 51 /r VSQRTPD ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 51 /r VSQRTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F	Computes Square Roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the square roots of the two, four or eight packed double-precision floating-point values in the source operand (the second operand) stores the packed double-precision floating-point results in the destination operand (the first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

**Operation****VSQRTPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND (SRC \*is register\*)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+63:i] ← SQRT(SRC[63:0])

ELSE DEST[i+63:i] ← SQRT(SRC[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VSQRTPD (VEX.256 encoded version)**

DEST[63:0] ← SQRT(SRC[63:0])

DEST[127:64] ← SQRT(SRC[127:64])

DEST[191:128] ← SQRT(SRC[191:128])

DEST[255:192] ← SQRT(SRC[255:192])

DEST[MAXVL-1:256] ← 0

**VSQRTPD (VEX.128 encoded version)**

DEST[63:0] ← SQRT(SRC[63:0])

DEST[127:64] ← SQRT(SRC[127:64])

DEST[MAXVL-1:128] ← 0

**SQRTPD (128-bit Legacy SSE version)**

DEST[63:0] ← SQRT(SRC[63:0])

DEST[127:64] ← SQRT(SRC[127:64])

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VSQRTPD \_\_m512d \_\_mm512\_sqrt\_round\_pd(\_\_m512d a, int r);

VSQRTPD \_\_m512d \_\_mm512\_mask\_sqrt\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int r);

VSQRTPD \_\_m512d \_\_mm512\_maskz\_sqrt\_round\_pd(\_\_mmask8 k, \_\_m512d a, int r);

VSQRTPD \_\_m256d \_\_mm256\_sqrt\_pd(\_\_m256d a);

VSQRTPD \_\_m256d \_\_mm256\_mask\_sqrt\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, int r);

VSQRTPD \_\_m256d \_\_mm256\_maskz\_sqrt\_pd(\_\_mmask8 k, \_\_m256d a, int r);

SQRTPD \_\_m128d \_\_mm\_sqrt\_pd(\_\_m128d a);

VSQRTPD \_\_m128d \_\_mm\_mask\_sqrt\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, int r);

VSQRTPD \_\_m128d \_\_mm\_maskz\_sqrt\_pd(\_\_mmask8 k, \_\_m128d a, int r);

### **SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

### **Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD                      If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD                      If EVEX.vvvv != 1111B.

## SQRTPS—Square Root of Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 51 /r SQRTPS xmm1, xmm2/m128	A	V/V	SSE	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.0F.WIG 51 /r VSQRTPS xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.0F.WIG 51/r VSQRTPS ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.0F.W0 51 /r VSQRTPS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 51 /r VSQRTPS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F	Computes Square Roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand.

EVEX.512 encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.



**Operation****VSQRTPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1) AND (SRC \*is register\*)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← SQRT(SRC[31:0])

ELSE DEST[i+31:i] ← SQRT(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VSQRTPS (VEX.256 encoded version)**

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

DEST[95:64] ← SQRT(SRC[95:64])

DEST[127:96] ← SQRT(SRC[127:96])

DEST[159:128] ← SQRT(SRC[159:128])

DEST[191:160] ← SQRT(SRC[191:160])

DEST[223:192] ← SQRT(SRC[223:192])

DEST[255:224] ← SQRT(SRC[255:224])

**VSQRTPS (VEX.128 encoded version)**

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

DEST[95:64] ← SQRT(SRC[95:64])

DEST[127:96] ← SQRT(SRC[127:96])

DEST[MAXVL-1:128] ← 0

**SQRTPS (128-bit Legacy SSE version)**

DEST[31:0] ← SQRT(SRC[31:0])

DEST[63:32] ← SQRT(SRC[63:32])

DEST[95:64] ← SQRT(SRC[95:64])

DEST[127:96] ← SQRT(SRC[127:96])

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSQRTPS __m512 _mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 _mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 _mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 _mm256_sqrt_ps(__m256 a);
VSQRTPS __m256 _mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 _mm_sqrt_ps(__m128 a);
VSQRTPS __m128 _mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 _mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD                    If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.

## SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51/r SQRTSD xmm1,xmm2/m64	A	V/V	SSE2	Computes square root of the low double-precision floating-point value in xmm2/m64 and stores the results in xmm1.
VEX.NDS.LIG.F2.0F.WIG 51/r VSQRTSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].
EVEX.NDS.LIG.F2.0F.W1 51/r VSQRTSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the square root of the low double-precision floating-point value in the second source operand and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSQRTSD (EVEX encoded version)**

```

IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SQRT(SRC2[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**VSQRTSD (VEX.128 encoded version)**

```

DEST[63:0] ← SQRT(SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**SQRTSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← SQRT(SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSQRTSD __m128d _mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.

## SQRTSS—Compute Square Root of Scalar Single-Precision Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS xmm1, xmm2/m32	A	V/V	SSE	Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.NDS.LIG.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].
EVEX.NDS.LIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the square root of the low single-precision floating-point value in the second source operand and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSQRTSS (EVEX encoded version)**

```

IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SQRT(SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] ← 0
        FI;
FI;
DEST[127:31] ← SRC1[127:31]
DEST[MAXVL-1:128] ← 0

```

**VSQRTSS (VEX.128 encoded version)**

```

DEST[31:0] ← SQRT(SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**SQRTSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← SQRT(SRC2[31:0])
DEST[MAXVL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_maskz_sqrt_round_ss(__mmask8 k, __m128 a, __m128 b, int r);
SQRTSS __m128 _mm_sqrt_ss(__m128 a)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 3.  
 EVEX-encoded instruction, see Exceptions Type E3.

## STAC—Set AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CB STAC	Z0	V/V	SMAP	Set the AC flag in the EFLAGS register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Sets the AC flag bit in EFLAGS register. This may enable alignment checking of user-mode data accesses. This allows explicit supervisor-mode data accesses to user-mode pages even if the SMAP bit is set in the CR4 register. This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute STAC when CPL > 0 cause #UD.

### Operation

EFLAGS.AC ← 1;

### Flags Affected

AC set. Other flags are unaffected.

### Protected Mode Exceptions

#UD  
     If the LOCK prefix is used.  
     If the CPL > 0.  
     If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### Real-Address Mode Exceptions

#UD  
     If the LOCK prefix is used.  
     If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### Virtual-8086 Mode Exceptions

#UD      The STAC instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD  
     If the LOCK prefix is used.  
     If the CPL > 0.  
     If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

### 64-Bit Mode Exceptions

#UD  
     If the LOCK prefix is used.  
     If the CPL > 0.  
     If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

## STC—Set Carry Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F9	STC	Z0	Valid	Valid	Set CF flag.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Sets the CF flag in the EFLAGS register. Operation is the same in all modes.

### Operation

$CF \leftarrow 1$ ;

### Flags Affected

The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

### Exceptions (All Operating Modes)

#UD                      If the LOCK prefix is used.



**STD—Set Direction Flag**

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
FD	STD	Z0	Valid	Valid	Set DF flag.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI). Operation is the same in all modes.

**Operation**

$DF \leftarrow 1;$

**Flags Affected**

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

**Exceptions (All Operating Modes)**

#UD If the LOCK prefix is used.

## STI—Set Interrupt Flag

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
FB	STI	Z0	Valid	Valid	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

In most cases, STI sets the interrupt flag (IF) in the EFLAGS register. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized<sup>1</sup>. If the STI instruction is followed by a CLI instruction (which clears the IF flag), the effect of the STI instruction is negated.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and NMI interrupts. NMI interrupts (and SMIs) may be blocked for one macroinstruction following an STI.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3, EFLAGS.VIP = 1, and either VME mode or PVI mode is active, STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-19 indicates the action of the STI instruction depending on the processor operating mode, IOPL, CPL, and EFLAGS.VIP.

**Table 4-19. Decision Table for STI Results**

Mode	IOPL	EFLAGS.VIP	STI Result
Real-address	X <sup>1</sup>	X	IF = 1
Protected, not PVI <sup>2</sup>	≥ CPL	X	IF = 1
	< CPL	X	#GP fault
Protected, PVI <sup>3</sup>	3	X	IF = 1
	0-2	0	VIF = 1
		1	#GP fault
Virtual-8086, not VME <sup>3</sup>	3	X	IF = 1
	0-2	X	#GP fault

1. The STI instruction delays recognition of interrupts only if it is executed with EFLAGS.IF = 0. In a sequence of STI instructions, only the first instruction in the sequence is guaranteed to delay interrupts.

In the following instruction sequence, interrupts may be recognized before RET executes:

```
STI
STI
RET
```

Table 4-19. Decision Table for STI Results

Mode	IOPL	EFLAGS.VIP	STI Result
Virtual-8086, VME <sup>3</sup>	3	X	IF = 1
	0-2	0	VIF = 1
		1	#GP fault

NOTES:

- 1. X = This setting has no effect on instruction operation.
- 2. For this table, “protected mode” applies whenever CRO.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
- 3. PVI mode and virtual-8086 mode each imply CPL = 3.

Operation

```
IF CRO.PE = 0 (* Executing in real-address mode *)
  THEN IF ← 1; (* Set Interrupt Flag *)
  ELSE
    IF IOPL ≥ CPL (* CPL = 3 if EFLAGS.VM = 1 *)
      THEN IF ← 1; (* Set Interrupt Flag *)
      ELSE
        IF VME mode OR PVI mode
          THEN
            IF EFLAGS.VIP = 0
              THEN VIF ← 1; (* Set Virtual Interrupt Flag *)
              ELSE #GP(0);
            FI;
          ELSE #GP(0);
        FI;
      FI;
    FI;
```

Flags Affected

Either the IF flag or the VIF flag is set to 1. Other flags are unaffected.

Protected Mode Exceptions

- #GP(0) If CPL is greater than IOPL and PVI mode is not active.
- If CPL is greater than IOPL and EFLAGS.VIP = 1.
- #UD If the LOCK prefix is used.

Real-Address Mode Exceptions

- #UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- #GP(0) If IOPL is less than 3 and VME mode is not active.
- If IOPL is less than 3 and EFLAGS.VIP = 1.
- #UD If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

## STMXCSR—Store MXCSR Register State

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF AE /3 STMXCSR <i>m32</i>	M	V/V	SSE	Store contents of MXCSR register to <i>m32</i> .
VEX.LZ.OF.WIG AE /3 VSTMXCSR <i>m32</i>	M	V/V	AVX	Store contents of MXCSR register to <i>m32</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

VEX.L must be 0, otherwise instructions will #UD.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

$m32 \leftarrow \text{MXCSR};$

### Intel C/C++ Compiler Intrinsic Equivalent

`_mm_getcsr(void)`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 5; additionally

#UD                      If VEX.L= 1,  
                            If VEX.vvvv ≠ 1111B.

## STOS/STOSB/STOSW/STOSD/STOSQ—Store String

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
AA	STOS <i>m8</i>	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOS <i>m16</i>	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOS <i>m32</i>	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOS <i>m64</i>	NA	Valid	N.E.	Store RAX at address RDI or EDI.
AA	STOSB	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOSW	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOSD	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOSQ	NA	Valid	N.E.	Store RAX at address RDI or EDI.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NA	NA	NA	NA	NA

## Description

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of the instruction are allowed: the “explicit-operands” form and the “no-operands” form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, EAX for doubleword operands). The explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI register. These must be loaded correctly before the store string instruction is executed.

The no-operands form provides “short forms” of the byte, word, doubleword, and quadword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and AL, AX, or EAX is assumed to be the source operand. The size of the destination and source operands is selected by the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the register to the memory location, the (E)DI register is incremented or decremented according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the register is incremented; if the DF flag is 1, the register is decremented (the register is incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for doubleword operations).

NOTE: To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with STOS and STOSB. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

In 64-bit mode, the default address size is 64 bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The promoted no-operand mnemonic is STOSQ. STOSQ (and its explicit operands variant) store a quadword from the RAX register into the destination addressed by RDI or EDI. See the summary chart at the beginning of this section for encoding data and limits.

The STOS, STOSB, STOSW, STOSD, STOSQ instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See "REP/REPE/REPZ/REPNE/REPZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

## Operation

Non-64-bit Mode:

```
IF (Byte store)
  THEN
    DEST ← AL;
    THEN IF DF = 0
      THEN (E)DI ← (E)DI + 1;
      ELSE (E)DI ← (E)DI - 1;
    FI;
  ELSE IF (Word store)
    THEN
      DEST ← AX;
      THEN IF DF = 0
        THEN (E)DI ← (E)DI + 2;
        ELSE (E)DI ← (E)DI - 2;
      FI;
    FI;
  ELSE IF (Doubleword store)
    THEN
      DEST ← EAX;
      THEN IF DF = 0
        THEN (E)DI ← (E)DI + 4;
        ELSE (E)DI ← (E)DI - 4;
      FI;
    FI;
  FI;
```

64-bit Mode:

```
IF (Byte store)
  THEN
    DEST ← AL;
    THEN IF DF = 0
      THEN (R)DI ← (R)DI + 1;
      ELSE (R)DI ← (R)DI - 1;
    FI;
  ELSE IF (Word store)
    THEN
      DEST ← AX;
```

```

        THEN IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 2;
            ELSE (R|E)DI ← (R|E)DI - 2;
        FI;
    FI;
ELSE IF (Doubleword store)
    THEN
        DEST ← EAX;
        THEN IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 4;
            ELSE (R|E)DI ← (R|E)DI - 4;
        FI;
    FI;
ELSE IF (Quadword store using REX.W )
    THEN
        DEST ← RAX;
        THEN IF DF = 0
            THEN (R|E)DI ← (R|E)DI + 8;
            ELSE (R|E)DI ← (R|E)DI - 8;
        FI;
    FI;
FI;

```

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	<p>If the destination is located in a non-writable segment.</p> <p>If a memory operand effective address is outside the limit of the ES segment.</p> <p>If the ES register contains a NULL segment selector.</p>
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the ES segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the ES segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.



**STR—Store Task Register**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /1	STR <i>r/m16</i>	M	Valid	Valid	Stores segment selector from TR in <i>r/m16</i> .

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> ( <i>w</i> )	NA	NA	NA

**Description**

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

In 64-bit mode, operation is the same. The size of the memory operand is fixed at 16 bits. In register stores, the 2-byte TR is zero extended if stored to a 64-bit register.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

**Operation**

DEST ← TR(SegmentSelector);

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If the destination is a memory operand that is located in a non-writable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector. If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD	The STR instruction is not recognized in real-address mode.
-----	---

**Virtual-8086 Mode Exceptions**

#UD	The STR instruction is not recognized in virtual-8086 mode.
-----	---

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## SUB—Subtract

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
2C <i>ib</i>	SUB AL, <i>imm8</i>	I	Valid	Valid	Subtract <i>imm8</i> from AL.
2D <i>iw</i>	SUB AX, <i>imm16</i>	I	Valid	Valid	Subtract <i>imm16</i> from AX.
2D <i>id</i>	SUB EAX, <i>imm32</i>	I	Valid	Valid	Subtract <i>imm32</i> from EAX.
REX.W + 2D <i>id</i>	SUB RAX, <i>imm32</i>	I	Valid	N.E.	Subtract <i>imm32</i> sign-extended to 64-bits from RAX.
80 /5 <i>ib</i>	SUB <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	Subtract <i>imm8</i> from <i>r/m8</i> .
REX + 80 /5 <i>ib</i>	SUB <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	Subtract <i>imm8</i> from <i>r/m8</i> .
81 /5 <i>iw</i>	SUB <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	Subtract <i>imm16</i> from <i>r/m16</i> .
81 /5 <i>id</i>	SUB <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	Subtract <i>imm32</i> from <i>r/m32</i> .
REX.W + 81 /5 <i>id</i>	SUB <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	Subtract <i>imm32</i> sign-extended to 64-bits from <i>r/m64</i> .
83 /5 <i>ib</i>	SUB <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	Subtract sign-extended <i>imm8</i> from <i>r/m16</i> .
83 /5 <i>ib</i>	SUB <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	Subtract sign-extended <i>imm8</i> from <i>r/m32</i> .
REX.W + 83 /5 <i>ib</i>	SUB <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	Subtract sign-extended <i>imm8</i> from <i>r/m64</i> .
28 /r	SUB <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Subtract <i>r8</i> from <i>r/m8</i> .
REX + 28 /r	SUB <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	Subtract <i>r8</i> from <i>r/m8</i> .
29 /r	SUB <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Subtract <i>r16</i> from <i>r/m16</i> .
29 /r	SUB <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Subtract <i>r32</i> from <i>r/m32</i> .
REX.W + 29 /r	SUB <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Subtract <i>r64</i> from <i>r/m64</i> .
2A /r	SUB <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Subtract <i>r/m8</i> from <i>r8</i> .
REX + 2A /r	SUB <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Subtract <i>r/m8</i> from <i>r8</i> .
2B /r	SUB <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Subtract <i>r/m16</i> from <i>r16</i> .
2B /r	SUB <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Subtract <i>r/m32</i> from <i>r32</i> .
REX.W + 2B /r	SUB <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Subtract <i>r/m64</i> from <i>r64</i> .

## NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

## Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	<i>imm8/26/32</i>	NA	NA
MI	ModRM: <i>r/m</i> ( <i>r</i> , <i>w</i> )	<i>imm8/26/32</i>	NA	NA
MR	ModRM: <i>r/m</i> ( <i>r</i> , <i>w</i> )	ModRM:reg ( <i>r</i> )	NA	NA
RM	ModRM:reg ( <i>r</i> , <i>w</i> )	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA

## Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

## Operation

$DEST \leftarrow (DEST - SRC);$

## Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## SUBPD—Subtract Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5C /r SUBPD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed double-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEEX.NDS.128.66.0F.WIG 5C /r VSUBPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed double-precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.
VEEX.NDS.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed double-precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.
EVEX.NDS.128.66.0F.W1 5C /r VSUBPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.66.0F.W1 5C /r VSUBPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.66.0F.W1 5C /r VSUBPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Subtract packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD subtract of the two, four or eight packed double-precision floating-point values of the second Source operand from the first Source operand, and stores the packed double-precision floating-point results in the destination operand.

VEEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation****VSUBPD (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VSUBPD (EVEX encoded versions) when src2 operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1)

THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[63:0];

ELSE EST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VSUBPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]

DEST[127:64] ← SRC1[127:64] - SRC2[127:64]

DEST[191:128] ← SRC1[191:128] - SRC2[191:128]

DEST[255:192] ← SRC1[255:192] - SRC2[255:192]

DEST[MAXVL-1:256] ← 0

**VSUBPD (VEX.128 encoded version)**

$$\text{DEST}[63:0] \leftarrow \text{SRC1}[63:0] - \text{SRC2}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] - \text{SRC2}[127:64]$$

$$\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$$
**SUBPD (128-bit Legacy SSE version)**

$$\text{DEST}[63:0] \leftarrow \text{DEST}[63:0] - \text{SRC}[63:0]$$

$$\text{DEST}[127:64] \leftarrow \text{DEST}[127:64] - \text{SRC}[127:64]$$

$$\text{DEST}[\text{MAXVL}-1:128] \text{ (Unmodified)}$$
**Intel C/C++ Compiler Intrinsic Equivalent**

$$\text{VSUBPD } \_m512d\_mm512\_sub\_pd (\_m512d\ a, \_m512d\ b);$$

$$\text{VSUBPD } \_m512d\_mm512\_mask\_sub\_pd (\_m512d\ s, \_mmask8\ k, \_m512d\ a, \_m512d\ b);$$

$$\text{VSUBPD } \_m512d\_mm512\_maskz\_sub\_pd (\_mmask8\ k, \_m512d\ a, \_m512d\ b);$$

$$\text{VSUBPD } \_m512d\_mm512\_sub\_round\_pd (\_m512d\ a, \_m512d\ b, \text{int});$$

$$\text{VSUBPD } \_m512d\_mm512\_mask\_sub\_round\_pd (\_m512d\ s, \_mmask8\ k, \_m512d\ a, \_m512d\ b, \text{int});$$

$$\text{VSUBPD } \_m512d\_mm512\_maskz\_sub\_round\_pd (\_mmask8\ k, \_m512d\ a, \_m512d\ b, \text{int});$$

$$\text{VSUBPD } \_m256d\_mm256\_sub\_pd (\_m256d\ a, \_m256d\ b);$$

$$\text{VSUBPD } \_m256d\_mm256\_mask\_sub\_pd (\_m256d\ s, \_mmask8\ k, \_m256d\ a, \_m256d\ b);$$

$$\text{VSUBPD } \_m256d\_mm256\_maskz\_sub\_pd (\_mmask8\ k, \_m256d\ a, \_m256d\ b);$$

$$\text{SUBPD } \_m128d\_mm\_sub\_pd (\_m128d\ a, \_m128d\ b);$$

$$\text{VSUBPD } \_m128d\_mm\_mask\_sub\_pd (\_m128d\ s, \_mmask8\ k, \_m128d\ a, \_m128d\ b);$$

$$\text{VSUBPD } \_m128d\_mm\_maskz\_sub\_pd (\_mmask8\ k, \_m128d\ a, \_m128d\ b);$$
**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5C /r SUBPS xmm1, xmm2/m128	A	V/V	SSE	Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 5C /r VSUBPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Subtract packed single-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.
VEX.NDS.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed single-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.
EVEX.NDS.128.0F.W0 5C /r VSUBPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.
EVEX.NDS.256.0F.W0 5C /r VSUBPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.
EVEX.NDS.512.0F.W0 5C /r VSUBPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F	Subtract packed single-precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD subtract of the packed single-precision floating-point values in the second Source operand from the First Source operand, and stores the packed single-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.



**Operation****VSUBPS (EVEX encoded versions) when src2 operand is a vector register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**VSUBPS (EVEX encoded versions) when src2 operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1)

THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[31:0];

ELSE DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**VSUBPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]

DEST[63:32] ← SRC1[63:32] - SRC2[63:32]

DEST[95:64] ← SRC1[95:64] - SRC2[95:64]

DEST[127:96] ← SRC1[127:96] - SRC2[127:96]

DEST[159:128] ← SRC1[159:128] - SRC2[159:128]

DEST[191:160] ← SRC1[191:160] - SRC2[191:160]

DEST[223:192] ← SRC1[223:192] - SRC2[223:192]

DEST[255:224] ← SRC1[255:224] - SRC2[255:224].

DEST[MAXVL-1:256] ← 0

**VSUBPS (VEX.128 encoded version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$

**SUBPS (128-bit Legacy SSE version)**

$\text{DEST}[31:0] \leftarrow \text{SRC1}[31:0] - \text{SRC2}[31:0]$   
 $\text{DEST}[63:32] \leftarrow \text{SRC1}[63:32] - \text{SRC2}[63:32]$   
 $\text{DEST}[95:64] \leftarrow \text{SRC1}[95:64] - \text{SRC2}[95:64]$   
 $\text{DEST}[127:96] \leftarrow \text{SRC1}[127:96] - \text{SRC2}[127:96]$   
 $\text{DEST}[\text{MAXVL}-1:128]$  (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

`VSUBPS __m512 __mm512_sub_ps (__m512 a, __m512 b);`  
`VSUBPS __m512 __mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);`  
`VSUBPS __m512 __mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);`  
`VSUBPS __m512 __mm512_sub_round_ps (__m512 a, __m512 b, int);`  
`VSUBPS __m512 __mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);`  
`VSUBPS __m512 __mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);`  
`VSUBPS __m256 __mm256_sub_ps (__m256 a, __m256 b);`  
`VSUBPS __m256 __mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);`  
`VSUBPS __m256 __mm256_maskz_sub_ps (__mmask8 k, __m256 a, __m256 b);`  
`SUBPS __m128 __mm_sub_ps (__m128 a, __m128 b);`  
`VSUBPS __m128 __mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);`  
`VSUBPS __m128 __mm_maskz_sub_ps (__mmask8 k, __m128 a, __m128 b);`

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

**SUBSD—Subtract Scalar Double-Precision Floating-Point Value**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5C /r SUBSD xmm1, xmm2/m64	A	V/V	SSE2	Subtract the low double-precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.
VEX.NDS.LIG.F2.0F.WIG 5C /r VSUBSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 5C /r VSUBSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Subtract the low double-precision floating-point value in the second source operand from the first source operand and stores the double-precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSUBSD (EVEX encoded version)**

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**VSUBSD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**SUBSD (128-bit Legacy SSE version)**

```

DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSUBSD __m128d __mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d __mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d __mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d __mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d __mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d __mm_sub_sd (__m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.  
 EVEX-encoded instructions, see Exceptions Type E3.

**SUBSS—Subtract Scalar Single-Precision Floating-Point Value**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS xmm1, xmm2/m32	A	V/V	SSE	Subtract the low single-precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.
VEX.NDS.LIG.F3.0F.WIG 5C /r VSUBSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 5C /r VSUBSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Subtract the low single-precision floating-point value from the second source operand and the first source operand and store the double-precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

**Operation****VSUBSS (EVEX encoded version)**

```

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**VSUBSS (VEX.128 encoded version)**

```

DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**SUBSS (128-bit Legacy SSE version)**

```

DEST[31:0] ← DEST[31:0] - SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.  
 EVEX-encoded instructions, see Exceptions Type E3.

## SWAPGS—Swap GS Base Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 01 F8	SWAPGS	Z0	Valid	Invalid	Exchanges the current GS base register value with the value contained in MSR address C0000102H.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32\_KERNEL\_GS\_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32\_KERNEL\_GS\_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The IA32\_KERNEL\_GS\_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32\_KERNEL\_GS\_BASE MSR contains a canonical address.

### Operation

IF CS.L  $\neq$  1 (\* Not in 64-Bit Mode \*)

THEN

#UD; FI;

IF CPL  $\neq$  0

THEN #GP(0); FI;

tmp  $\leftarrow$  GS.base;

GS.base  $\leftarrow$  IA32\_KERNEL\_GS\_BASE;

IA32\_KERNEL\_GS\_BASE  $\leftarrow$  tmp;

### Flags Affected

None

### Protected Mode Exceptions

#UD If Mode  $\neq$  64-Bit.

### Real-Address Mode Exceptions

#UD If Mode  $\neq$  64-Bit.

### Virtual-8086 Mode Exceptions

#UD If Mode  $\neq$  64-Bit.

**Compatibility Mode Exceptions**

#UD                      If Mode  $\neq$  64-Bit.

**64-Bit Mode Exceptions**

#GP(0)                If CPL  $\neq$  0.

#UD                      If the LOCK prefix is used.



## SYSCALL—Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 05	SYSCALL	Z0	Valid	Invalid	Fast call to privilege level 0 system procedures.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32\_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32\_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32\_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32\_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32\_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

### Operation

IF (CS.L  $\neq$  1) or (IA32\_EFER.LMA  $\neq$  1) or (IA32\_EFER.SCE  $\neq$  1)  
 (\* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32\_EFER \*)

THEN #UD;

FI;

RCX  $\leftarrow$  RIP; (\* Will contain address of next instruction \*)

RIP  $\leftarrow$  IA32\_LSTAR;

R11  $\leftarrow$  RFLAGS;

RFLAGS  $\leftarrow$  RFLAGS AND NOT(IA32\_FMASK);

CS.Selector  $\leftarrow$  IA32\_STAR[47:32] AND FFFCH (\* Operating system provides CS; RPL forced to 0 \*)

(\* Set rest of CS to a fixed value \*)

CS.Base  $\leftarrow$  0; (\* Flat segment \*)

CS.Limit  $\leftarrow$  FFFFFFFH; (\* With 4-KByte granularity, implies a 4-GByte limit \*)

CS.Type  $\leftarrow$  11; (\* Execute/read code, accessed \*)

CS.S  $\leftarrow$  1;

CS.DPL  $\leftarrow$  0;

CS.P  $\leftarrow$  1;

CS.L  $\leftarrow$  1; (\* Entry is to 64-bit mode \*)

CS.D  $\leftarrow$  0; (\* Required if CS.L = 1 \*)

CS.G  $\leftarrow$  1; (\* 4-KByte granularity \*)

CPL  $\leftarrow$  0;

SS.Selector $\leftarrow$ IA32_STAR[47:32] + 8;	(* SS just above CS *)
(* Set rest of SS to a fixed value *)	
SS.Base $\leftarrow$ 0;	(* Flat segment *)
SS.Limit $\leftarrow$ FFFFFFFH;	(* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type $\leftarrow$ 3;	(* Read/write data, accessed *)
SS.S $\leftarrow$ 1;	
SS.DPL $\leftarrow$ 0;	
SS.P $\leftarrow$ 1;	
SS.B $\leftarrow$ 1;	(* 32-bit stack segment *)
SS.G $\leftarrow$ 1;	(* 4-KByte granularity *)

### Flags Affected

All.

### Protected Mode Exceptions

#UD The SYSCALL instruction is not recognized in protected mode.

### Real-Address Mode Exceptions

#UD The SYSCALL instruction is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD The SYSCALL instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD The SYSCALL instruction is not recognized in compatibility mode.

### 64-Bit Mode Exceptions

#UD If IA32\_EFER.SCE = 0.  
If the LOCK prefix is used.

## SYSENTER—Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 34	SYSENTER	Z0	Valid	Valid	Fast call to privilege level 0 system procedures.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32\_SYSENTER\_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32\_SYSENTER\_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32\_SYSENTER\_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32\_SYSENTER\_EIP and IA32\_SYSENTER\_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32\_SYSENTER\_CS MSR, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER\_CS\_MSR MSR.
- The fast system call “stub” routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

### Operation

```
IF CRO.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); FI;

RFLAGS.VM ← 0;                (* Ensures protected mode execution *)
RFLAGS.IF ← 0;                (* Mask interrupts *)
IF in IA-32e mode
    THEN
        RSP ← IA32_SYSENTER_ESP;
        RIP ← IA32_SYSENTER_EIP;
    ELSE
        ESP ← IA32_SYSENTER_ESP[31:0];
        EIP ← IA32_SYSENTER_EIP[31:0];
FI;

CS.Selector ← IA32_SYSENTER_CS[15:0] AND FFFCH;
                                (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                   (* Flat segment *)
CS.Limit ← FFFFFFFH;           (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                  (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
IF in IA-32e mode
    THEN
        CS.L ← 1;              (* Entry is to 64-bit mode *)
        CS.D ← 0;              (* Required if CS.L = 1 *)
    ELSE
        CS.L ← 0;
        CS.D ← 1;              (* 32-bit code segment *)
FI;
CS.G ← 1;                      (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← CS.Selector + 8;  (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                   (* Flat segment *)
SS.Limit ← FFFFFFFH;           (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                   (* Read/write data, accessed *)
```

```
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1; (* 32-bit stack segment*)
SS.G ← 1; (* 4-KByte granularity *)
```

**Flags Affected**

VM, IF (see Operation above)

**Protected Mode Exceptions**

- #GP(0) If IA32\_SYSENTER\_CS[15:2] = 0.
- #UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

- #GP The SYSENTER instruction is not recognized in real-address mode.
- #UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 35	SYSEXIT	Z0	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	Z0	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protection levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32\_SYSENTER\_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **ECX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32\_SYSENTER\_CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32\_SYSENTER\_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
    FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

**Operation**

IF IA32\_SYSENTER\_CS[15:2] = 0 OR CRO.PE = 0 OR CPL ≠ 0 THEN #GP(0); FI;

IF operand size is 64-bit

THEN (\* Return to 64-bit mode \*)

RSP ← RCX;

RIP ← RDX;

ELSE (\* Return to protected mode or compatibility mode \*)

RSP ← ECX;

RIP ← EDX;

FI;

IF operand size is 64-bit (\* Operating system provides CS; RPL forced to 3 \*)

THEN CS.Selector ← IA32\_SYSENTER\_CS[15:0] + 32;

ELSE CS.Selector ← IA32\_SYSENTER\_CS[15:0] + 16;

FI;

CS.Selector ← CS.Selector OR 3; (\* RPL forced to 3 \*)

(\* Set rest of CS to a fixed value \*)

CS.Base ← 0; (\* Flat segment \*)

CS.Limit ← FFFFFFFH; (\* With 4-KByte granularity, implies a 4-GByte limit \*)

CS.Type ← 11; (\* Execute/read code, accessed \*)

CS.S ← 1;

CS.DPL ← 3;

CS.P ← 1;

IF operand size is 64-bit

THEN (\* return to 64-bit mode \*)

CS.L ← 1; (\* 64-bit code segment \*)

CS.D ← 0; (\* Required if CS.L = 1 \*)

ELSE (\* return to protected mode or compatibility mode \*)

CS.L ← 0;

CS.D ← 1; (\* 32-bit code segment\*)

FI;

CS.G ← 1; (\* 4-KByte granularity \*)

CPL ← 3;

SS.Selector ← CS.Selector + 8; (\* SS just above CS \*)

(\* Set rest of SS to a fixed value \*)

SS.Base ← 0; (\* Flat segment \*)

SS.Limit ← FFFFFFFH; (\* With 4-KByte granularity, implies a 4-GByte limit \*)

SS.Type ← 3; (\* Read/write data, accessed \*)

SS.S ← 1;

SS.DPL ← 3;

SS.P ← 1;

SS.B ← 1; (\* 32-bit stack segment\*)

SS.G ← 1; (\* 4-KByte granularity \*)

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0) If IA32\_SYSENTER\_CS[15:2] = 0.

If CPL ≠ 0.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	The SYSEXIT instruction is not recognized in real-address mode.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	The SYSEXIT instruction is not recognized in virtual-8086 mode.
--------	---

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	If IA32_SYSENTER_CS = 0. If CPL $\neq$ 0. If RCX or RDX contains a non-canonical address.
#UD	If the LOCK prefix is used.



## SYSRET—Return From Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 07	SYSRET	Z0	Valid	Invalid	Return to compatibility mode from fast system call
REX.W + 0F 07	SYSRET	Z0	Valid	Invalid	Return to 64-bit mode from fast system call

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.<sup>1</sup> With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32\_STAR MSR. However, the CS and SS descriptor caches are not loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- Nonmaskable interrupts (NMIs). The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, “Interrupt Stack Table,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).
- General-protection exceptions (#GP). The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - Confirming that the value of RCX is canonical before executing SYSRET.
  - Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - Using the IST mechanism for gate 13 (#GP) in the IDT.

### Operation

IF (CS.L  $\neq$  1) or (IA32\_EFER.LMA  $\neq$  1) or (IA32\_EFER.SCE  $\neq$  1)  
 (\* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32\_EFER \*)  
 THEN #UD; FI;  
 IF (CPL  $\neq$  0) OR (RCX is not canonical) THEN #GP(0); FI;

1. Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.

```

IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP ← ECX;
FI;
RFLAGS ← (R11 & 3C7FD7H) | 2;          (* Clear RF, VM, reserved bits; set bit 2 *)

IF (operand size is 64-bit)
    THEN CS.Selector ← IA32_STAR[63:48]+16;
    ELSE CS.Selector ← IA32_STAR[63:48];
FI;
CS.Selector ← CS.Selector OR 3;          (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                            (* Flat segment *)
CS.Limit ← FFFFFFFH;                    (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                           (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        CS.L ← 1;                        (* 64-bit code segment *)
        CS.D ← 0;                        (* Required if CS.L = 1 *)
    ELSE (* Return to Compatibility Mode *)
        CS.L ← 0;                        (* Compatibility mode *)
        CS.D ← 1;                        (* 32-bit code segment *)
FI;
CS.G ← 1;                               (* 4-KByte granularity *)
CPL ← 3;

SS.Selector ← (IA32_STAR[63:48]+8) OR 3; (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                            (* Flat segment *)
SS.Limit ← FFFFFFFH;                    (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                            (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                               (* 32-bit stack segment *)
SS.G ← 1;                               (* 4-KByte granularity *)

```

### Flags Affected

All.

### Protected Mode Exceptions

#UD The SYSRET instruction is not recognized in protected mode.

### Real-Address Mode Exceptions

#UD The SYSRET instruction is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD The SYSRET instruction is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD                      The SYSRET instruction is not recognized in compatibility mode.

### 64-Bit Mode Exceptions

#UD                      If IA32\_EFER.SCE = 0.  
                            If the LOCK prefix is used.

#GP(0)                  If CPL ≠ 0.  
                            If RCX contains a non-canonical address.

## TEST—Logical Compare

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
A8 <i>ib</i>	TEST AL, <i>imm8</i>	I	Valid	Valid	AND <i>imm8</i> with AL; set SF, ZF, PF according to result.
A9 <i>iw</i>	TEST AX, <i>imm16</i>	I	Valid	Valid	AND <i>imm16</i> with AX; set SF, ZF, PF according to result.
A9 <i>id</i>	TEST EAX, <i>imm32</i>	I	Valid	Valid	AND <i>imm32</i> with EAX; set SF, ZF, PF according to result.
REX.W + A9 <i>id</i>	TEST RAX, <i>imm32</i>	I	Valid	N.E.	AND <i>imm32</i> sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.
F6 /0 <i>ib</i>	TEST <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	AND <i>imm8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
REX + F6 /0 <i>ib</i>	TEST <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	AND <i>imm8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
F7 /0 <i>iw</i>	TEST <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	AND <i>imm16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result.
F7 /0 <i>id</i>	TEST <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	AND <i>imm32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result.
REX.W + F7 /0 <i>id</i>	TEST <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	AND <i>imm32</i> sign-extended to 64-bits with <i>r/m64</i> ; set SF, ZF, PF according to result.
84 / <i>r</i>	TEST <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	AND <i>r8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
REX + 84 / <i>r</i>	TEST <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	AND <i>r8</i> with <i>r/m8</i> ; set SF, ZF, PF according to result.
85 / <i>r</i>	TEST <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	AND <i>r16</i> with <i>r/m16</i> ; set SF, ZF, PF according to result.
85 / <i>r</i>	TEST <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	AND <i>r32</i> with <i>r/m32</i> ; set SF, ZF, PF according to result.
REX.W + 85 / <i>r</i>	TEST <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	AND <i>r64</i> with <i>r/m64</i> ; set SF, ZF, PF according to result.

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	<i>imm8/16/32</i>	NA	NA
MI	ModRM: <i>r/m</i> ( <i>r</i> )	<i>imm8/16/32</i>	NA	NA
MR	ModRM: <i>r/m</i> ( <i>r</i> )	ModRM:reg ( <i>r</i> )	NA	NA

### Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

TEMP  $\leftarrow$  SRC1 AND SRC2;  
 SF  $\leftarrow$  MSB(TEMP);

IF TEMP = 0  
     THEN ZF  $\leftarrow$  1;  
     ELSE ZF  $\leftarrow$  0;

FI:

PF  $\leftarrow$  BitwiseXNOR(TEMP[0:7]);  
 CF  $\leftarrow$  0;  
 OF  $\leftarrow$  0;  
 (\* AF is undefined \*)

**Flags Affected**

The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the “Operation” section above). The state of the AF flag is undefined.

**Protected Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## TZCNT — Count the Number of Trailing Zero Bits

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 0F BC /r TZCNT <i>r16, r/m16</i>	A	V/V	BMI1	Count the number of trailing zero bits in <i>r/m16</i> , return result in <i>r16</i> .
F3 0F BC /r TZCNT <i>r32, r/m32</i>	A	V/V	BMI1	Count the number of trailing zero bits in <i>r/m32</i> , return result in <i>r32</i> .
F3 REX.W 0F BC /r TZCNT <i>r64, r/m64</i>	A	V/N.E.	BMI1	Count the number of trailing zero bits in <i>r/m64</i> , return result in <i>r64</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between TZCNT and BSF instruction is that TZCNT provides operand size as output when source operand is zero while in the case of BSF instruction, if source operand is zero, the content of destination operand are undefined. On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

### Operation

```
temp ← 0
DEST ← 0
DO WHILE ( (temp < OperandSize) and (SRC[ temp] = 0) )
```

```
    temp ← temp + 1
    DEST ← DEST + 1
OD
```

```
IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI
```

```
IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI
```

### Flags Affected

ZF is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

### Intel C/C++ Compiler Intrinsic Equivalent

```
TZCNT:    unsigned __int32 _tzcnt_u32(unsigned __int32 src);
TZCNT:    unsigned __int64 _tzcnt_u64(unsigned __int64 src);
```

**Protected Mode Exceptions**

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Real-Address Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.

**Virtual 8086 Mode Exceptions**

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

**Compatibility Mode Exceptions**

Same exceptions as in Protected Mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2E /r UCOMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Performs an unordered compare of the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid numeric exception only if a source operand is either an SNaN or a QNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### (V)UCOMISD (all versions)

```
RESULT ← UnorderedCompare(DEST[63:0] <> SRC[63:0]) {
```

```
(* Set EFLAGS *) CASE (RESULT) OF
```

```
  UNORDERED: ZF,PF,CF ← 111;
```

```
  GREATER_THAN: ZF,PF,CF ← 000;
```

```
  LESS_THAN: ZF,PF,CF ← 001;
```

```
  EQUAL: ZF,PF,CF ← 100;
```

```
ESAC;
```

```
OF, AF, SF ← 0; }
```



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VUCOMISD int __mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
UCOMISD int __mm_ucomieq_sd(__m128d a, __m128d b)
UCOMISD int __mm_ucomilt_sd(__m128d a, __m128d b)
UCOMISD int __mm_ucomile_sd(__m128d a, __m128d b)
UCOMISD int __mm_ucomigt_sd(__m128d a, __m128d b)
UCOMISD int __mm_ucomige_sd(__m128d a, __m128d b)
UCOMISD int __mm_ucomineq_sd(__m128d a, __m128d b)

```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN operands), Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD                      If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 2E /r UCOMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.OF.WIG 2E /r VUCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.OF.WO 2E /r VUCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

### Operation

#### (V)UCOMISS (all versions)

```
RESULT ← UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
```

```
(* Set EFLAGS *) CASE (RESULT) OF
```

```
    UNORDERED: ZF,PF,CF ← 111;
```

```
    GREATER_THAN: ZF,PF,CF ← 000;
```

```
    LESS_THAN: ZF,PF,CF ← 001;
```

```
    EQUAL: ZF,PF,CF ← 100;
```

```
ESAC;
```

```
OF, AF, SF ← 0; }
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VUCOMISS  int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
UCOMISS   int _mm_ucomieq_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomilt_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomile_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomigt_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomige_ss(__m128 a, __m128 b);
UCOMISS   int _mm_ucomineq_ss(__m128 a, __m128 b);

```

**SIMD Floating-Point Exceptions**

Invalid (if SNaN Operands), Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD                      If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

## UD—Undefined Instruction

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F FF /r	<sup>1</sup> <i>r32, r/m32</i>	RM	Valid	Valid	Raise invalid opcode exception.
0F B9 /r	UD1 <i>r32, r/m32</i>	RM	Valid	Valid	Raise invalid opcode exception.
0F 0B	UD2	ZO	Valid	Valid	Raise invalid opcode exception.

### NOTES:

1. Some older processors decode the UD0 instruction without a ModR/M byte. As a result, those processors would deliver an invalid-opcode exception instead of a fault on instruction fetch when the instruction with a ModR/M byte (and any implied bytes) would cross a page or segment boundary.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

### Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcodes for this instruction are reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD instruction (and not the following instruction).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

#UD (\* Generates invalid opcode exception \*);

### Flags Affected

None.

### Exceptions (All Operating Modes)

#UD                      Raises an invalid opcode exception in all operating modes.

## UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 15 /r UNPCKHPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 15 /r UNPCKHPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 15 /r UNPCKHPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.66.0F.W1 15 /r UNPCKHPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 15 /r UNPCKHPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 15 /r UNPCKHPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Unpacks and Interleaves double-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an interleaved unpack of the high double-precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

## Operation

### VUNPCKHPD (EVEX encoded versions when SRC2 is a register)

$$(KL, VL) = (2, 128), (4, 256), (8, 512)$$

IF VL  $\geq$  128

TMP\_DEST[63:0]  $\leftarrow$  SRC1[127:64]

TMP\_DEST[127:64] ← SRC2[127:64]

Fl;

IF VL >= 256

TMP\_DEST[191:128] ← SRC1[255:192]

TMP\_DEST[255:192] ← SRC2[255:192]

Fl;

IF VL >= 512

TMP\_DEST[319:256] ← SRC1[383:320]

TMP\_DEST[383:320] ← SRC2[383:320]

TMP\_DEST[447:384] ← SRC1[511:448]

TMP\_DEST[511:448] ← SRC2[511:448]

Fl;

FOR j  $\leftarrow$  0 TO KL-1

$$i \leftarrow j * 64$$

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i]  $\leftarrow$  TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

$$\text{DEST}[i+63:i] \leftarrow 0$$

FI

Fl;

ENDFOR

$$\text{DEST}[\text{MAXVL}-1:\text{VL}] \leftarrow 0$$

**VUNPCKHPD (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1)

THEN TMP\_SRC2[i+63:i] ← SRC2[63:0]

ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i]

FI;

ENDFOR;

IF VL &gt;= 128

TMP\_DEST[63:0] ← SRC1[127:64]

TMP\_DEST[127:64] ← TMP\_SRC2[127:64]

FI;

IF VL &gt;= 256

TMP\_DEST[191:128] ← SRC1[255:192]

TMP\_DEST[255:192] ← TMP\_SRC2[255:192]

FI;

IF VL &gt;= 512

TMP\_DEST[319:256] ← SRC1[383:320]

TMP\_DEST[383:320] ← TMP\_SRC2[383:320]

TMP\_DEST[447:384] ← SRC1[511:448]

TMP\_DEST[511:448] ← TMP\_SRC2[511:448]

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VUNPCKHPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[127:64]

DEST[127:64] ← SRC2[127:64]

DEST[191:128] ← SRC1[255:192]

DEST[255:192] ← SRC2[255:192]

DEST[MAXVL-1:256] ← 0

**VUNPCKHPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[127:64]

DEST[127:64] ← SRC2[127:64]

DEST[MAXVL-1:128] ← 0

**UNPCKHPD (128-bit Legacy SSE version)**

DEST[63:0] ← SRC1[127:64]

DEST[127:64] ← SRC2[127:64]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VUNPCKHPD __m512d _mm512_unpackhi_pd( __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_mask_unpackhi_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_maskz_unpackhi_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b)
VUNPCKHPD __m256d _mm256_mask_unpackhi_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_maskz_unpackhi_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)
VUNPCKHPD __m128d _mm_mask_unpackhi_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKHPD __m128d _mm_maskz_unpackhi_pd(__mmask8 k, __m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.



## UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 15 /r UNPCKHPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.OF.WIG 15 /r VUNPCKHPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.OF.WIG 15 /r VUNPCKHPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.OF.W0 15 /r VUNPCKHPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.
EVEX.NDS.256.OF.W0 15 /r VUNPCKHPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.
EVEX.NDS.512.OF.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

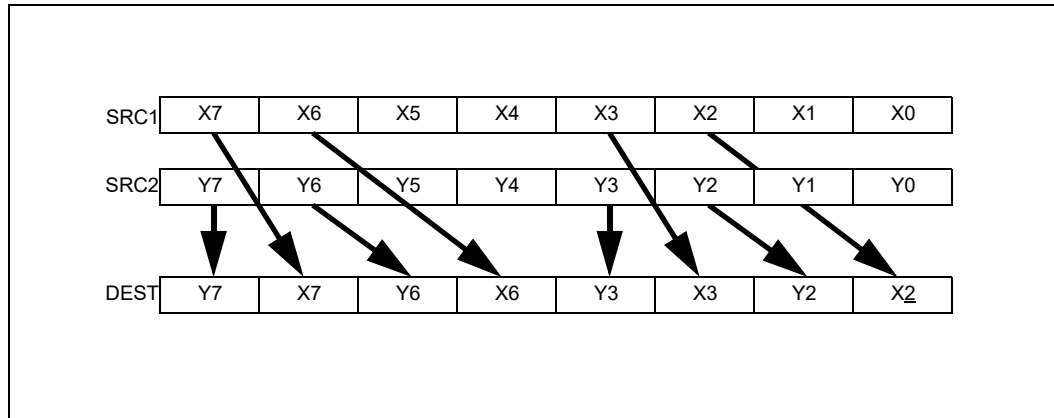


Figure 4-27. VUNPCKHPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKHPS (EVEX encoded version when SRC2 is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL >= 128

```
TMP_DEST[31:0] ← SRC1[95:64]
TMP_DEST[63:32] ← SRC2[95:64]
TMP_DEST[95:64] ← SRC1[127:96]
TMP_DEST[127:96] ← SRC2[127:96]
```

FI;

IF VL >= 256

```
TMP_DEST[159:128] ← SRC1[223:192]
TMP_DEST[191:160] ← SRC2[223:192]
TMP_DEST[223:192] ← SRC1[255:224]
TMP_DEST[255:224] ← SRC2[255:224]
```

FI;

IF VL >= 512

```
TMP_DEST[287:256] ← SRC1[351:320]
TMP_DEST[319:288] ← SRC2[351:320]
TMP_DEST[351:320] ← SRC1[383:352]
TMP_DEST[383:352] ← SRC2[383:352]
TMP_DEST[415:384] ← SRC1[479:448]
TMP_DEST[447:416] ← SRC2[479:448]
TMP_DEST[479:448] ← SRC1[511:480]
TMP_DEST[511:480] ← SRC2[511:480]
```

FI;

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VUNPCKHPS (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL >= 128
  TMP_DEST[31:0] ← SRC1[95:64]
  TMP_DEST[63:32] ← TMP_SRC2[95:64]
  TMP_DEST[95:64] ← SRC1[127:96]
  TMP_DEST[127:96] ← TMP_SRC2[127:96]
FI;
IF VL >= 256
  TMP_DEST[159:128] ← SRC1[223:192]
  TMP_DEST[191:160] ← TMP_SRC2[223:192]
  TMP_DEST[223:192] ← SRC1[255:224]
  TMP_DEST[255:224] ← TMP_SRC2[255:224]
FI;
IF VL >= 512
  TMP_DEST[287:256] ← SRC1[351:320]
  TMP_DEST[319:288] ← TMP_SRC2[351:320]
  TMP_DEST[351:320] ← SRC1[383:352]
  TMP_DEST[383:352] ← TMP_SRC2[383:352]
  TMP_DEST[415:384] ← SRC1[479:448]
  TMP_DEST[447:416] ← TMP_SRC2[479:448]
  TMP_DEST[479:448] ← SRC1[511:480]
  TMP_DEST[511:480] ← TMP_SRC2[511:480]
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR

```

FI

FI;

ENDFOR

DEST[MAXVL-1:VL]  $\leftarrow$  0**VUNPCKHPS (VEX.256 encoded version)**DEST[31:0]  $\leftarrow$  SRC1[95:64]DEST[63:32]  $\leftarrow$  SRC2[95:64]DEST[95:64]  $\leftarrow$  SRC1[127:96]DEST[127:96]  $\leftarrow$  SRC2[127:96]DEST[159:128]  $\leftarrow$  SRC1[223:192]DEST[191:160]  $\leftarrow$  SRC2[223:192]DEST[223:192]  $\leftarrow$  SRC1[255:224]DEST[255:224]  $\leftarrow$  SRC2[255:224]DEST[MAXVL-1:256]  $\leftarrow$  0**VUNPCKHPS (VEX.128 encoded version)**DEST[31:0]  $\leftarrow$  SRC1[95:64]DEST[63:32]  $\leftarrow$  SRC2[95:64]DEST[95:64]  $\leftarrow$  SRC1[127:96]DEST[127:96]  $\leftarrow$  SRC2[127:96]DEST[MAXVL-1:128]  $\leftarrow$  0**UNPCKHPS (128-bit Legacy SSE version)**DEST[31:0]  $\leftarrow$  SRC1[95:64]DEST[63:32]  $\leftarrow$  SRC2[95:64]DEST[95:64]  $\leftarrow$  SRC1[127:96]DEST[127:96]  $\leftarrow$  SRC2[127:96]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VUNPCKHPS \_\_m512 \_\_mm512\_unpackhi\_ps( \_\_m512 a, \_\_m512 b);

VUNPCKHPS \_\_m512 \_\_mm512\_mask\_unpackhi\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);

VUNPCKHPS \_\_m512 \_\_mm512\_maskz\_unpackhi\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);

VUNPCKHPS \_\_m256 \_\_mm256\_unpackhi\_ps( \_\_m256 a, \_\_m256 b);

VUNPCKHPS \_\_m256 \_\_mm256\_mask\_unpackhi\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);

VUNPCKHPS \_\_m256 \_\_mm256\_maskz\_unpackhi\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256 b);

UNPCKHPS \_\_m128 \_\_mm\_unpackhi\_ps( \_\_m128 a, \_\_m128 b);

VUNPCKHPS \_\_m128 \_\_mm\_mask\_unpackhi\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VUNPCKHPS \_\_m128 \_\_mm\_maskz\_unpackhi\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

## UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 14 /r UNPCKLPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 14 /r VUNPCKLPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 14 /r VUNPCKLPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.66.0F.W1 14 /r VUNPCKLPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128/m64bcst subject to write mask k1.
EVEX.NDS.256.66.0F.W1 14 /r VUNPCKLPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256/m64bcst subject to write mask k1.
EVEX.NDS.512.66.0F.W1 14 /r VUNPCKLPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Unpacks and Interleaves double-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an interleaved unpack of the low double-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or a 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

**Operation****VUNPCKLPD (EVEX encoded versions when SRC2 is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL &gt;= 128

TMP\_DEST[63:0] ← SRC1[63:0]

TMP\_DEST[127:64] ← SRC2[63:0]

FI;

IF VL &gt;= 256

TMP\_DEST[191:128] ← SRC1[191:128]

TMP\_DEST[255:192] ← SRC2[191:128]

FI;

IF VL &gt;= 512

TMP\_DEST[319:256] ← SRC1[319:256]

TMP\_DEST[383:320] ← SRC2[319:256]

TMP\_DEST[447:384] ← SRC1[447:384]

TMP\_DEST[511:448] ← SRC2[447:384]

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VUNPCKLPD (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[127:64] ← TMP_SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[191:128]
    TMP_DEST[255:192] ← TMP_SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[319:256]
    TMP_DEST[383:320] ← TMP_SRC2[319:256]
    TMP_DEST[447:384] ← SRC1[447:384]
    TMP_DEST[511:448] ← TMP_SRC2[447:384]
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VUNPCKLPD (VEX.256 encoded version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[191:128] ← SRC1[191:128]
DEST[255:192] ← SRC2[191:128]
DEST[MAXVL-1:256] ← 0

```

**VUNPCKLPD (VEX.128 encoded version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAXVL-1:128] ← 0

```

**UNPCKLPD (128-bit Legacy SSE version)**

```

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VUNPCKLPD __m512d _mm512_unpacklo_pd( __m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_mask_unpacklo_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_maskz_unpacklo_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d b)
VUNPCKLPD __m256d _mm256_mask_unpacklo_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKLPD __m256d _mm256_maskz_unpacklo_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKLPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)
VUNPCKLPD __m128d _mm_mask_unpacklo_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKLPD __m128d _mm_maskz_unpacklo_pd(__mmask8 k, __m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.



## UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF 14 /r UNPCKLPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.OF.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.OF.WIG 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.OF.W0 14 /r VUNPCKLPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.
EVEX.NDS.256.OF.W0 14 /r VUNPCKLPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.
EVEX.NDS.512.OF.W0 14 /r VUNPCKLPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

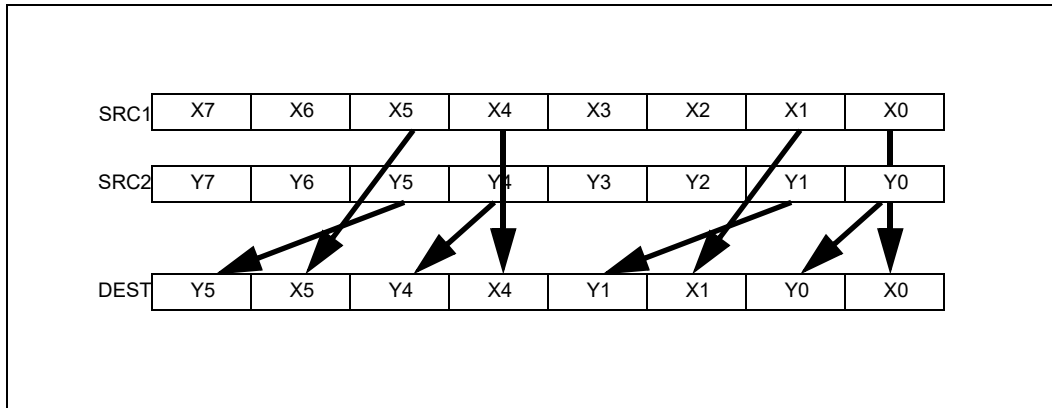


Figure 4-28. VUNPCKLPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

### Operation

#### VUNPCKLPS (EVEX encoded version when SRC2 is a ZMM register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL ≥ 128

```
TMP_DEST[31:0] ← SRC1[31:0]
TMP_DEST[63:32] ← SRC2[31:0]
TMP_DEST[95:64] ← SRC1[63:32]
TMP_DEST[127:96] ← SRC2[63:32]
```

FI;

IF VL ≥ 256

```
TMP_DEST[159:128] ← SRC1[159:128]
TMP_DEST[191:160] ← SRC2[159:128]
TMP_DEST[223:192] ← SRC1[191:160]
TMP_DEST[255:224] ← SRC2[191:160]
```

FI;

IF VL ≥ 512

```
TMP_DEST[287:256] ← SRC1[287:256]
TMP_DEST[319:288] ← SRC2[287:256]
TMP_DEST[351:320] ← SRC1[319:288]
TMP_DEST[383:352] ← SRC2[319:288]
TMP_DEST[415:384] ← SRC1[415:384]
TMP_DEST[447:416] ← SRC2[415:384]
TMP_DEST[479:448] ← SRC1[447:416]
TMP_DEST[511:480] ← SRC2[447:416]
```

FI;

FOR j ← 0 TO KL-1

  i ← j \* 32

```

    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VUNPCKLPS (EVEX encoded version when SRC2 is memory)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 31
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[31:0]
    TMP_DEST[63:32] ← TMP_SRC2[31:0]
    TMP_DEST[95:64] ← SRC1[63:32]
    TMP_DEST[127:96] ← TMP_SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[159:128]
    TMP_DEST[191:160] ← TMP_SRC2[159:128]
    TMP_DEST[223:192] ← SRC1[191:160]
    TMP_DEST[255:224] ← TMP_SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[287:256]
    TMP_DEST[319:288] ← TMP_SRC2[287:256]
    TMP_DEST[351:320] ← SRC1[319:288]
    TMP_DEST[383:352] ← TMP_SRC2[319:288]
    TMP_DEST[415:384] ← SRC1[415:384]
    TMP_DEST[447:416] ← TMP_SRC2[415:384]
    TMP_DEST[479:448] ← SRC1[447:416]
    TMP_DEST[511:480] ← TMP_SRC2[447:416]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR

```

ENDFOR  
 DEST[MAXVL-1:VL] ← 0

#### UNPCKLPS (VEX.256 encoded version)

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[159:128] ← SRC1[159:128]  
 DEST[191:160] ← SRC2[159:128]  
 DEST[223:192] ← SRC1[191:160]  
 DEST[255:224] ← SRC2[191:160]  
 DEST[MAXVL-1:256] ← 0

#### VUNPCKLPS (VEX.128 encoded version)

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[MAXVL-1:128] ← 0

#### UNPCKLPS (128-bit Legacy SSE version)

DEST[31:0] ← SRC1[31:0]  
 DEST[63:32] ← SRC2[31:0]  
 DEST[95:64] ← SRC1[63:32]  
 DEST[127:96] ← SRC2[63:32]  
 DEST[MAXVL-1:128] (Unmodified)

#### Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPS \_\_m512 \_mm512\_unpacklo\_ps(\_\_m512 a, \_\_m512 b);  
 VUNPCKLPS \_\_m512 \_mm512\_mask\_unpacklo\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VUNPCKLPS \_\_m512 \_mm512\_maskz\_unpacklo\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512 b);  
 VUNPCKLPS \_\_m256 \_mm256\_unpacklo\_ps (\_\_m256 a, \_\_m256 b);  
 VUNPCKLPS \_\_m256 \_mm256\_mask\_unpacklo\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 VUNPCKLPS \_\_m256 \_mm256\_maskz\_unpacklo\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256 b);  
 UNPCKLPS \_\_m128 \_mm\_unpacklo\_ps (\_\_m128 a, \_\_m128 b);  
 VUNPCKLPS \_\_m128 \_mm\_mask\_unpacklo\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);  
 VUNPCKLPS \_\_m128 \_mm\_maskz\_unpacklo\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128 b);

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4NF.

## VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 03 /r ib VALIGND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Shift right and merge vectors xmm2 and xmm3/m128/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.NDS.128.66.0F3A.W1 03 /r ib VALIGNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Shift right and merge vectors xmm2 and xmm3/m128/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W0 03 /r ib VALIGND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Shift right and merge vectors ymm2 and ymm3/m256/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.NDS.256.66.0F3A.W1 03 /r ib VALIGNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Shift right and merge vectors ymm2 and ymm3/m256/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W0 03 /r ib VALIGND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Shift right and merge vectors zmm2 and zmm3/m512/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.
EVEX.NDS.512.66.0F3A.W1 03 /r ib VALIGNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Shift right and merge vectors zmm2 and zmm3/m512/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand) into a 1024/512/256-bit intermediate vector. The low 512/256/128-bit of the intermediate vector is written to the destination operand (the first operand) using the writemask k1. The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values (merging-masking) or are set to 0 (zeroing-masking).

**Operation****VALIGND (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
        FOR j ← 0 TO KL-1
            i ← j * 32
            src[i+31:i] ← SRC2[31:0]
        ENDFOR;
    ELSE src ← SRC2
FI
; Concatenate sources
tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]
; Shift right doubleword elements
IF VL = 128
    THEN SHIFT = imm8[1:0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[2:0]
            ELSE SHIFT = imm8[3:0]
        FI
FI;
tmp[2VL-1:0] ← tmp[2VL-1:0] >> (32*SHIFT)
; Apply writemask
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← tmp[i+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**VALIGNQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (SRC2 \*is memory\*) (AND EVEX.b = 1)

THEN

FOR j  $\leftarrow$  0 TO KL-1i  $\leftarrow$  j \* 64src[j+63:i]  $\leftarrow$  SRC2[63:0]

ENDFOR;

ELSE src  $\leftarrow$  SRC2

FI

; Concatenate sources

tmp[VL-1:0]  $\leftarrow$  src[VL-1:0]tmp[2VL-1:VL]  $\leftarrow$  SRC1[VL-1:0]

; Shift right quadword elements

IF VL = 128

THEN SHIFT = imm8[0]

ELSE

IF VL = 256

THEN SHIFT = imm8[1:0]

ELSE SHIFT = imm8[2:0]

FI

FI;

tmp[2VL-1:0]  $\leftarrow$  tmp[2VL-1:0]  $\gg$  (64\*SHIFT)

; Apply writemask

FOR j  $\leftarrow$  0 TO KL-1i  $\leftarrow$  j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i]  $\leftarrow$  tmp[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i]  $\leftarrow$  0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VALIGND __m512i_mm512_alignr_epi32(__m512i a, __m512i b, int cnt);
VALIGND __m512i_mm512_mask_alignr_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i_mm512_maskz_alignr_epi32(__mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i_mm256_mask_alignr_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i_mm256_maskz_alignr_epi32(__mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i_mm_mask_alignr_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i_mm_maskz_alignr_epi32(__mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m512i_mm512_alignr_epi64(__m512i a, __m512i b, int cnt);
VALIGNQ __m512i_mm512_mask_alignr_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i_mm512_maskz_alignr_epi64(__mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i_mm256_mask_alignr_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i_mm256_maskz_alignr_epi64(__mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i_mm_mask_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i_mm_maskz_alignr_epi64(__mmask8 k, __m128i a, __m128i b, int cnt);

```

**Exceptions**

See Exceptions Type E4NF.



## VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 65 /r VBLENDMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512F	Blend double-precision vector xmm2 and double-precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W1 65 /r VBLENDMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL AVX512F	Blend double-precision vector ymm2 and double-precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W1 65 /r VBLENDMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F	Blend double-precision vector zmm2 and double-precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.
EVEX.NDS.128.66.0F38.W0 65 /r VBLENDMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512F	Blend single-precision vector xmm2 and single-precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W0 65 /r VBLENDMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512F	Blend single-precision vector ymm2 and single-precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W0 65 /r VBLENDMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F	Blend single-precision vector zmm2 and single-precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source operand, 1 for second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

**Operation****VBLENDMPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no controlmask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] ← SRC2[63:0]

ELSE

DEST[i+63:i] ← SRC2[i+63:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN DEST[i+63:i] ← SRC1[i+63:i]

ELSE

; zeroing-masking

DEST[i+63:i] ← 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VBLENDMPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no controlmask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN DEST[i+31:i] ← SRC1[i+31:i]

ELSE

; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VBLENDMPD __m512d _mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);  
VBLENDMPD __m256d _mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);  
VBLENDMPD __m128d _mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);  
VBLENDMPS __m512 _mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);  
VBLENDMPS __m256 _mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);  
VBLENDMPS __m128 _mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

**VBROADCAST—Load with Broadcast Floating-Point Data**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, m64	A	V/V	AVX	Broadcast double-precision floating-point element in mem to four locations in ymm1.
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 ymm1, m128	A	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.
VEX.128.66.0F38.W0 18/r VBROADCASTSS xmm1, xmm2	A	V/V	AVX2	Broadcast the low single-precision floating-point element in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, xmm2	A	V/V	AVX2	Broadcast low single-precision floating-point element in the source operand to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, xmm2	A	V/V	AVX2	Broadcast low double-precision floating-point element in the source operand to four locations in ymm1.
EVEX.256.66.0F38.W1 19 /r VBROADCASTSD ymm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast low double-precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 19 /r VBROADCASTSD zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F	Broadcast low double-precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 19 /r VBROADCASTF32X2 ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two single-precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 19 /r VBROADCASTF32X2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ	Broadcast two single-precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 18 /r VBROADCASTSS zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 1A /r VBROADCASTF32X4 ymm1 {k1}{z}, m128	D	V/V	AVX512VL AVX512F	Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1A /r VBROADCASTF32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F	Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 1A /r VBROADCASTF64X2 ymm1 {k1}{z}, m128	C	V/V	AVX512VL AVX512DQ	Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in ymm1 using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 1A /r VBROADCASTF64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ	Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 1B /r VBROADCASTF32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ	Broadcast 256 bits of 8 single-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 1B /r VBROADCASTF64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F	Broadcast 256 bits of 4 double-precision floating-point data in mem to locations in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

VEX256-encoded versions: The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded versions: The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF32X8/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

VBROADCASTSD and VBROADCASTF128,F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. VBROADCASTSS is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF32X8 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

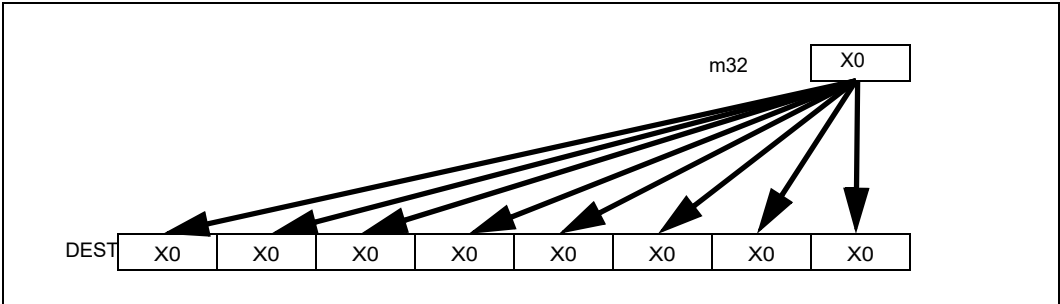


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

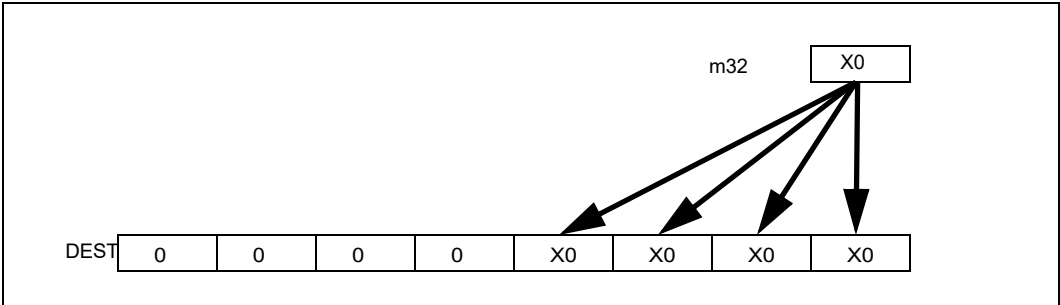


Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

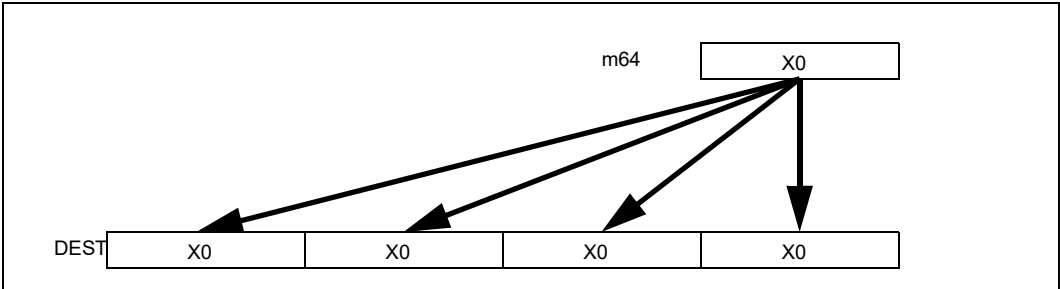


Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

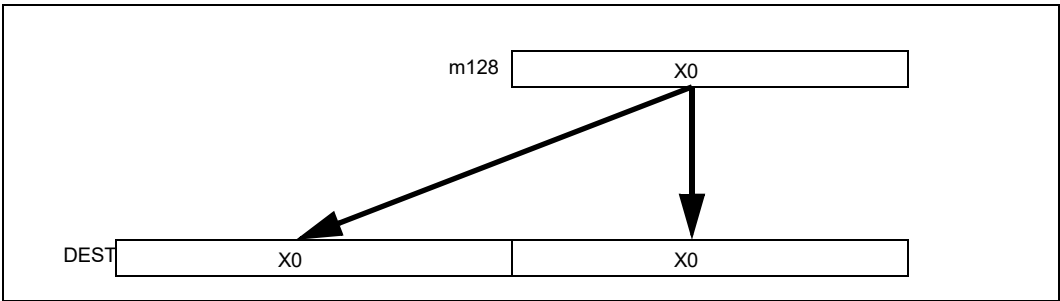


Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)

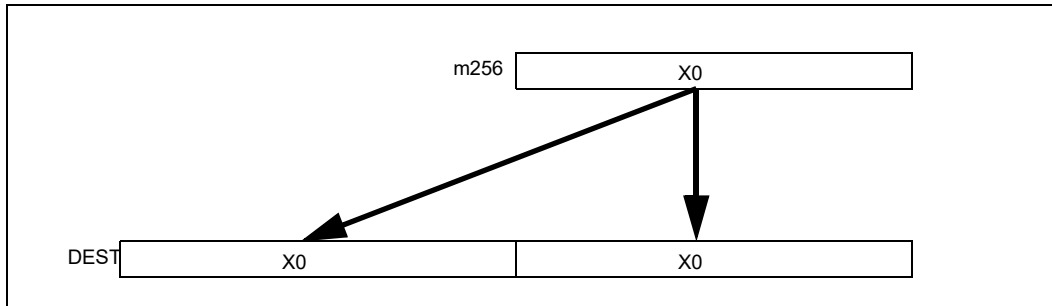


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

### Operation

#### VBROADCASTSS (128 bit version VEX and legacy)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[MAXVL-1:128] ← 0
```

#### VBROADCASTSS (VEX.256 encoded version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
DEST[MAXVL-1:256] ← 0
```

#### VBROADCASTSS (EVEX encoded versions)

(KL, VL) (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← SRC[31:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VBROADCASTSD (VEX.256 encoded version)**

```

temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAXVL-1:256] ← 0

```

**VBROADCASTSD (EVEX encoded versions)**

```

(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VBROADCASTF32x2 (EVEX encoded versions)**

```

(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    n ← (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VBROADCASTF128 (VEX.256 encoded version)**

```

temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAXVL-1:256] ← 0

```



**VBROADCASTF32X4 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    n ← (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VBROADCASTF64X2 (EVEX encoded versions)**

(KL, VL) = (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    n ← (j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+63:i] = 0
        FI
    FI;
ENDFOR;

```

**VBROADCASTF32X8 (EVEX.U1.512 encoded version)**

FOR j ← 0 TO 15

```

    i ← j * 32
    n ← (j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VBROADCASTF64X4 (EVEX.512 encoded version)**

```

FOR j ← 0 TO 7
  i ← j * 64
  n ← (j modulo 4) * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[n+63:n]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VBROADCASTF32x2 __m512 _mm512_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m512 _mm512_mask_broadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskz_broadcast_f32x2( __mmask16 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m256 _mm256_mask_broadcast_f32x2(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_maskz_broadcast_f32x2( __mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m512 _mm512_mask_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskz_broadcast_f32x4( __mmask16 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m256 _mm256_mask_broadcast_f32x4(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_maskz_broadcast_f32x4( __mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_broadcast_f32x8( __m256 a);
VBROADCASTF32x8 __m512 _mm512_mask_broadcast_f32x8(__m512 s, __mmask16 k, __m256 a);
VBROADCASTF32x8 __m512 _mm512_maskz_broadcast_f32x8( __mmask16 k, __m256 a);
VBROADCASTF64x2 __m512d _mm512_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m512d _mm512_mask_broadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d _mm512_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m256d _mm256_mask_broadcast_f64x2(__m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d _mm512_broadcast_f64x4( __m256d a);
VBROADCASTF64x4 __m512d _mm512_mask_broadcast_f64x4(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512d _mm512_maskz_broadcast_f64x4( __mmask8 k, __m256d a);
VBROADCASTSD __m512d _mm512_broadcastsd_pd( __m128d a);
VBROADCASTSD __m512d _mm512_mask_broadcastsd_pd(__m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcastsd_pd(__m128d a);
VBROADCASTSD __m256d _mm256_mask_broadcastsd_pd(__m256d s, __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcast_sd(double *a);
VBROADCASTSS __m512 _mm512_broadcastss_ps( __m128 a);
VBROADCASTSS __m512 _mm512_mask_broadcastss_ps(__m512 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512 _mm512_maskz_broadcastss_ps( __mmask16 k, __m128 a);
VBROADCASTSS __m256 _mm256_broadcastss_ps(__m128 a);
VBROADCASTSS __m256 _mm256_mask_broadcast_ss(__m256 s, __mmask8 k, __m128 a);
VBROADCASTSS __m256 _mm256_maskz_broadcast_ss( __mmask8 k, __m128 a);

```

```

VBROADCASTSS __m128 _mm_broadcastss_ps(__m128 a);
VBROADCASTSS __m128 _mm_mask_broadcast_ss(__m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_maskz_broadcast_ss(__mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_broadcast_ss(float *a);
VBROADCASTSS __m256 _mm256_broadcast_ss(float *a);
VBROADCASTF128 __m256 _mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d _mm256_broadcast_pd(__m128d * a);

```

### Exceptions

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6.

#UD                      If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.  
                           If EVEX.L'L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.  
                           If EVEX.L'L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.

## VCOMPRESSPD—Store Sparse Packed Double-Precision Floating-Point Values into Dense Memory

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8A /r VCOMPRESSPD xmm1/m128 {k1}{z}, xmm2	A	V/V	AVX512VL AVX512F	Compress packed double-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W1 8A /r VCOMPRESSPD ymm1/m256 {k1}{z}, ymm2	A	V/V	AVX512VL AVX512F	Compress packed double-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W1 8A /r VCOMPRESSPD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F	Compress packed double-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Compress (store) up to 8 double-precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VCOMPRESSPD (EVEX encoded versions) store form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN

            DEST[k+SIZE-1:k] ← SRC[i+63:i]

            k ← k + SIZE

    FI;

ENDFOR

**VCOMPRESSPD (EVEX encoded versions) reg-reg form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+63:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCOMPRESSPD \_\_m512d \_\_mm512\_mask\_compress\_pd( \_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VCOMPRESSPD \_\_m512d \_\_mm512\_maskz\_compress\_pd( \_\_mmask8 k, \_\_m512d a);

VCOMPRESSPD void \_\_mm512\_mask\_compressstoreu\_pd( void \* d, \_\_mmask8 k, \_\_m512d a);

VCOMPRESSPD \_\_m256d \_\_mm256\_mask\_compress\_pd( \_\_m256d s, \_\_mmask8 k, \_\_m256d a);

VCOMPRESSPD \_\_m256d \_\_mm256\_maskz\_compress\_pd( \_\_mmask8 k, \_\_m256d a);

VCOMPRESSPD void \_\_mm256\_mask\_compressstoreu\_pd( void \* d, \_\_mmask8 k, \_\_m256d a);

VCOMPRESSPD \_\_m128d \_\_mm\_mask\_compress\_pd( \_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VCOMPRESSPD \_\_m128d \_\_mm\_maskz\_compress\_pd( \_\_mmask8 k, \_\_m128d a);

VCOMPRESSPD void \_\_mm\_mask\_compressstoreu\_pd( void \* d, \_\_mmask8 k, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.

**VCOMPRESSPS—Store Sparse Packed Single-Precision Floating-Point Values into Dense Memory**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8A /r VCOMPRESSPS xmm1/m128 {k1}{z}, xmm2	A	V/V	AVX512VL AVX512F	Compress packed single-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W0 8A /r VCOMPRESSPS ymm1/m256 {k1}{z}, ymm2	A	V/V	AVX512VL AVX512F	Compress packed single-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W0 8A /r VCOMPRESSPS zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F	Compress packed single-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

**Description**

Compress (stores) up to 16 single-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (a partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation****VCOMPRESSPS (EVEX encoded versions) store form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[k+SIZE-1:k] ← SRC[i+31:i]

      k ← k + SIZE

  FI;

ENDFOR;

**VCOMPRESSPS (EVEX encoded versions) reg-reg form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE  $\leftarrow$  32k  $\leftarrow$  0FOR j  $\leftarrow$  0 TO KL-1i  $\leftarrow$  j \* 32

IF k1[j] OR \*no writemask\*

THEN

DEST[k+SIZE-1:k]  $\leftarrow$  SRC[i+31:i]k  $\leftarrow$  k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k]  $\leftarrow$  0

FI

DEST[MAXVL-1:VL]  $\leftarrow$  0**Intel C/C++ Compiler Intrinsic Equivalent**

VCOMPRESSPS \_\_m512 \_\_mm512\_mask\_compress\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VCOMPRESSPS \_\_m512 \_\_mm512\_maskz\_compress\_ps( \_\_mmask16 k, \_\_m512 a);

VCOMPRESSPS void \_\_mm512\_mask\_compressstoreu\_ps( void \* d, \_\_mmask16 k, \_\_m512 a);

VCOMPRESSPS \_\_m256 \_\_mm256\_mask\_compress\_ps( \_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VCOMPRESSPS \_\_m256 \_\_mm256\_maskz\_compress\_ps( \_\_mmask8 k, \_\_m256 a);

VCOMPRESSPS void \_\_mm256\_mask\_compressstoreu\_ps( void \* d, \_\_mmask8 k, \_\_m256 a);

VCOMPRESSPS \_\_m128 \_\_mm\_mask\_compress\_ps( \_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VCOMPRESSPS \_\_m128 \_\_mm\_maskz\_compress\_ps( \_\_mmask8 k, \_\_m128 a);

VCOMPRESSPS void \_\_mm\_mask\_compressstoreu\_ps( void \* d, \_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD If EVEX.vvvv  $\neq$  1111B.

## VCVTPD2QQ—Convert Packed Double-Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7B /r VCVTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 7B /r VCVTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 7B /r VCVTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where w represents the number of bits in the destination format) is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.



**Operation****VCVTPD2QQ (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_QuadInteger(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTPD2QQ (EVEX encoded version) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b == 1)

THEN

DEST[i+63:i] ← Convert\_Double\_Precision\_Floating\_Point\_To\_QuadInteger(SRC[63:0])

ELSE

DEST[i+63:i] ← Convert\_Double\_Precision\_Floating\_Point\_To\_QuadInteger(SRC[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPD2QQ __m512i __mm512_cvtpd_epi64( __m512d a);
VCVTPD2QQ __m512i __mm512_mask_cvtpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i __mm512_maskz_cvtpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m512i __mm512_cvt_roundpd_epi64( __m512d a, int r);
VCVTPD2QQ __m512i __mm512_mask_cvt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m512i __mm512_maskz_cvt_roundpd_epi64( __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m256i __mm256_mask_cvtpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i __mm256_maskz_cvtpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ __m128i __mm_mask_cvtpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2QQ __m128i __mm_maskz_cvtpd_epi64( __mmask8 k, __m128d a);
VCVTPD2QQ __m256i __mm256_cvtpd_epi64( __m256d src)
VCVTPD2QQ __m128i __mm_cvtpd_epi64( __m128d src)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2

#UD                      If EVEX.vvvv != 1111B.

## VCVTPD2UDQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 subject to writemask k1.
EVEX.256.0F.W1 79 /r VCVTPD2UDQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in ymm1 subject to writemask k1.
EVEX.512.0F.W1 79 /r VCVTPD2UDQ zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VCVTPD2UDQ (EVEX encoded versions) when src2 operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0

**VCVTPD2UDQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[63:0])

ELSE

DEST[i+31:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[k+63:k])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPD2UDQ __m256i _mm512_cvtpd_epu32( __m512d a);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_maskz_cvtpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_cvt_roundpd_epu32( __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_mask_cvt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_maskz_cvt_roundpd_epu32( __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i _mm256_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm256_maskz_cvtpd_epu32( __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i _mm_maskz_cvtpd_epu32( __mmask8 k, __m128d a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD                      If EVEX.vvvv != 1111B.

## VCVTPD2UQQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 79 /r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 79 /r VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert fourth packed double-precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 79 /r VCVTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VCVTPD2UQQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTPD2UQQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b == 1)

THEN

DEST[i+63:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger(SRC[63:0])

ELSE

DEST[i+63:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger(SRC[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPD2UQQ __m512i _mm512_cvtpd_epu64( __m512d a);
VCVTPD2UQQ __m512i _mm512_mask_cvtpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_maskz_cvtpd_epu64( __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_cvt_roundpd_epu64( __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_mask_cvt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_maskz_cvt_roundpd_epu64( __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i _mm256_mask_cvtpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i _mm256_maskz_cvtpd_epu64( __mmask8 k, __m256d a);
VCVTPD2UQQ __m128i _mm_mask_cvtpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i _mm_maskz_cvtpd_epu64( __mmask8 k, __m128d a);
VCVTPD2UQQ __m256i _mm256_cvtpd_epu64( __m256d src)
VCVTPD2UQQ __m128i _mm_cvtpd_epu64( __m128d src)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2

#UD                      If EVEX.vvvv != 1111B.



## VCVTPH2PS—Convert 16-bit FP values to Single-Precision FP values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64	A	V/V	F16C	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.
VEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128	A	V/V	F16C	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.
EVEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point values in xmm1.
EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point values in ymm1.
EVEX.512.66.0F38.W0 13 /r VCVTPH2PS zmm1 {k1}{z}, ymm2/m256 {sae}	B	V/V	AVX512F	Convert sixteen packed half precision (16-bit) floating-point values in ymm2/m256 to packed single-precision floating-point values in zmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed half precision (16-bits) floating-point values in the low-order bits of the source operand (the second operand) to packed single-precision floating-point values and writes the converted values into the destination operand (the first operand).

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) FP values.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

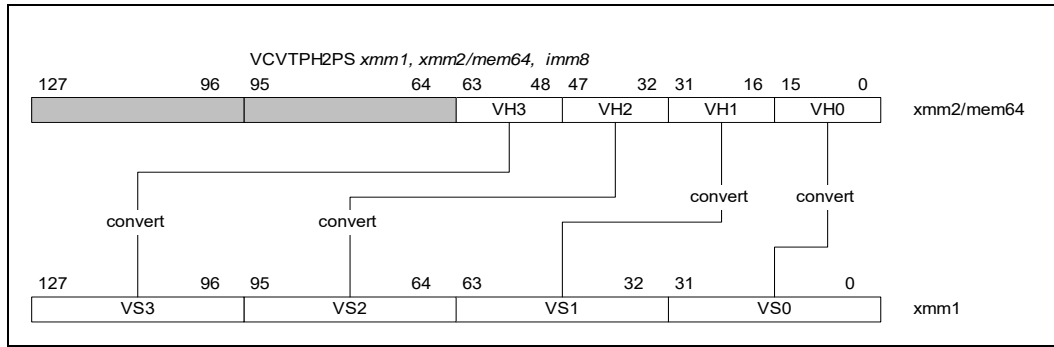


Figure 5-6. VCVTPH2PS (128-bit Version)

**Operation**

```

vCvt_h2s(SRC1[15:0])
{
  RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}

```

**VCVTPH2PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  k ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ←

      vCvt\_h2s(SRC[k+15:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTPH2PS (VEX.256 encoded version)**

DEST[31:0] ← vCvt\_h2s(SRC1[15:0]);

DEST[63:32] ← vCvt\_h2s(SRC1[31:16]);

DEST[95:64] ← vCvt\_h2s(SRC1[47:32]);

DEST[127:96] ← vCvt\_h2s(SRC1[63:48]);

DEST[159:128] ← vCvt\_h2s(SRC1[79:64]);

DEST[191:160] ← vCvt\_h2s(SRC1[95:80]);

DEST[223:192] ← vCvt\_h2s(SRC1[111:96]);

DEST[255:224] ← vCvt\_h2s(SRC1[127:112]);

DEST[MAXVL-1:256] ← 0

**VCVTPH2PS (VEX.128 encoded version)**

$DEST[31:0] \leftarrow vCvt\_h2s(SRC1[15:0]);$   
 $DEST[63:32] \leftarrow vCvt\_h2s(SRC1[31:16]);$   
 $DEST[95:64] \leftarrow vCvt\_h2s(SRC1[47:32]);$   
 $DEST[127:96] \leftarrow vCvt\_h2s(SRC1[63:48]);$   
 $DEST[MAXVL-1:128] \leftarrow 0$

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

`VCVTPH2PS __m512 __mm512_cvtph_ps( __m256i a);`  
`VCVTPH2PS __m512 __mm512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);`  
`VCVTPH2PS __m512 __mm512_maskz_cvtph_ps(__mmask16 k, __m256i a);`  
`VCVTPH2PS __m512 __mm512_cvt_roundph_ps( __m256i a, int sae);`  
`VCVTPH2PS __m512 __mm512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);`  
`VCVTPH2PS __m512 __mm512_maskz_cvt_roundph_ps( __mmask16 k, __m256i a, int sae);`  
`VCVTPH2PS __m256 __mm256_mask_cvtph_ps(__m256 s, __mmask8 k, __m128i a);`  
`VCVTPH2PS __m256 __mm256_maskz_cvtph_ps(__mmask8 k, __m128i a);`  
`VCVTPH2PS __m128 __mm_mask_cvtph_ps(__m128 s, __mmask8 k, __m128i a);`  
`VCVTPH2PS __m128 __mm_maskz_cvtph_ps(__mmask8 k, __m128i a);`  
`VCVTPH2PS __m128 __mm_cvtph_ps( __m128i m1);`  
`VCVTPH2PS __m256 __mm256_cvtph_ps( __m128i m1)`

**SIMD Floating-Point Exceptions**

Invalid

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);

EVEX-encoded instructions, see Exceptions Type E11.

#UD If VEX.W=1.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m64, xmm2, imm8	A	V/V	F16C	Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
VEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m128, ymm2, imm8	A	V/V	F16C	Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.128.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m64 {k1}{z}, xmm2, imm8	B	V/V	AVX512VL AVX512F	Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
EVEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	AVX512VL AVX512F	Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.512.66.0F3A.W0 1D /r ib VCVTPS2PH ymm1/m256 {k1}{z}, zmm2{sae}, imm8	B	V/V	AVX512F	Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

Convert packed single-precision floating values in the source operand to half-precision (16-bit) floating-point values and store to the destination operand. The rounding mode is specified using the immediate field (imm8).

Underflow results (i.e., tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to the input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

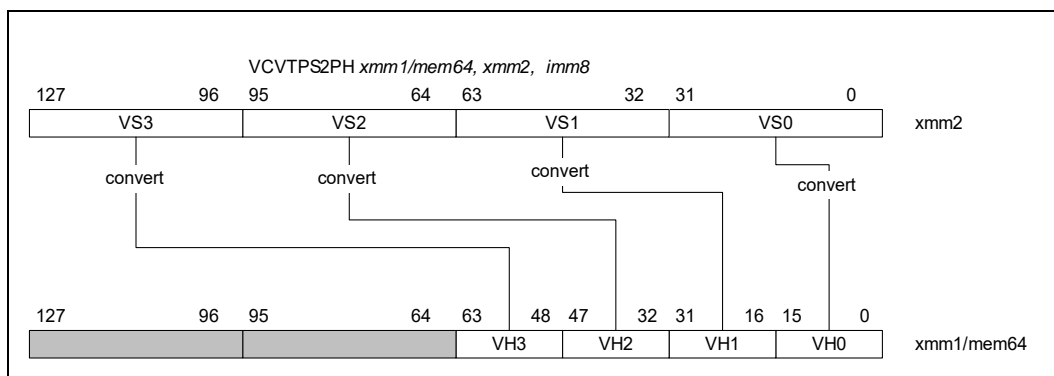


Figure 5-7. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 5-12.

**Table 5-12. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions**

Bits	Field Name/value	Description	Comment
Imm[1:0]	RC=00B	Round to nearest even	If Imm[2] = 0
	RC=01B	Round down	
	RC=10B	Round up	
	RC=11B	Truncate	
Imm[2]	MS1=0	Use imm[1:0] for rounding	Ignore MXCSR.RC
	MS1=1	Use MXCSR.RC for rounding	
Imm[7:3]	Ignored	Ignored by processor	

VEX.128 version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If the destination operand is a register then the upper bits (MAXVL-1:64) of corresponding register are zeroed.

VEX.256 version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

EVEX encoded versions: The source operand is a ZMM/ymm/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAXVL-1:256/128/64) of the corresponding destination register are zeroed.

### Operation

```

vCvt_s2h(SRC1[31:0])
{
  IF Imm[2] = 0
  THEN   ; using Imm[1:0] for rounding control, see Table 5-12
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
  ELSE   ; using MXCSR.RC for rounding control
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
  FI;
}
```

### VCVTPS2PH (EVEX encoded versions) when dest is a register

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ←
      vCvt_s2h(SRC[k+31:k])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE                           ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

**VCVTPS2PH (EVEX encoded versions) when dest is memory**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 16
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ←
            vCvt_s2h(SRC[k+31:k])
    ELSE
        *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR

```

**VCVTPS2PH (VEX.256 encoded version)**

```

DEST[15:0] ← vCvt_s2h(SRC1[31:0]);
DEST[31:16] ← vCvt_s2h(SRC1[63:32]);
DEST[47:32] ← vCvt_s2h(SRC1[95:64]);
DEST[63:48] ← vCvt_s2h(SRC1[127:96]);
DEST[79:64] ← vCvt_s2h(SRC1[159:128]);
DEST[95:80] ← vCvt_s2h(SRC1[191:160]);
DEST[111:96] ← vCvt_s2h(SRC1[223:192]);
DEST[127:112] ← vCvt_s2h(SRC1[255:224]);
DEST[MAXVL-1:128] ← 0

```

**VCVTPS2PH (VEX.128 encoded version)**

```

DEST[15:0] ← vCvt_s2h(SRC1[31:0]);
DEST[31:16] ← vCvt_s2h(SRC1[63:32]);
DEST[47:32] ← vCvt_s2h(SRC1[95:64]);
DEST[63:48] ← vCvt_s2h(SRC1[127:96]);
DEST[MAXVL-1:64] ← 0

```

**Flags Affected**

None

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPS2PH __m256i __mm512_cvtps_ph(__m512 a);
VCVTPS2PH __m256i __mm512_mask_cvtps_ph(__m256i s, __mmask16 k, __m512 a);
VCVTPS2PH __m256i __mm512_maskz_cvtps_ph(__mmask16 k, __m512 a);
VCVTPS2PH __m256i __mm512_cvt_roundps_ph(__m512 a, const int imm);
VCVTPS2PH __m256i __mm512_mask_cvt_roundps_ph(__m256i s, __mmask16 k, __m512 a, const int imm);
VCVTPS2PH __m256i __mm512_maskz_cvt_roundps_ph(__mmask16 k, __m512 a, const int imm);
VCVTPS2PH __m128i __mm256_mask_cvtps_ph(__m128i s, __mmask8 k, __m256 a);
VCVTPS2PH __m128i __mm256_maskz_cvtps_ph(__mmask8 k, __m256 a);
VCVTPS2PH __m128i __mm_mask_cvtps_ph(__m128i s, __mmask8 k, __m128 a);
VCVTPS2PH __m128i __mm_maskz_cvtps_ph(__mmask8 k, __m128 a);
VCVTPS2PH __m128i __mm_cvtps_ph (__m128 m1, const int imm);
VCVTPS2PH __m128i __mm256_cvtps_ph(__m256 m1, const int imm);

```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);

EVEX-encoded instructions, see Exceptions Type E11.

#UD                      If VEX.W=1.

#UD                      If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VCVTPS2UDQ—Convert Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W0 79 /r VCVTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 subject to writemask k1.
EVEX.256.0F.W0 79 /r VCVTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 subject to writemask k1.
EVEX.512.0F.W0 79 /r VCVTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	A	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts sixteen packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.



**Operation****VCVTPS2UDQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTPS2UDQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[31:0])

ELSE

DEST[i+31:i] ←

Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPS2UDQ __m512i __mm512_cvtps_epu32( __m512 a);
VCVTPS2UDQ __m512i __mm512_mask_cvtps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i __mm512_maskz_cvtps_epu32( __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i __mm512_cvt_roundps_epu32( __m512 a, int r);
VCVTPS2UDQ __m512i __mm512_mask_cvt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i __mm512_maskz_cvt_roundps_epu32( __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i __mm256_cvtps_epu32( __m256d a);
VCVTPS2UDQ __m256i __mm256_mask_cvtps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i __mm256_maskz_cvtps_epu32( __mmask8 k, __m256 a);
VCVTPS2UDQ __m128i __mm_cvtps_epu32( __m128 a);
VCVTPS2UDQ __m128i __mm_mask_cvtps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i __mm_maskz_cvtps_epu32( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD                      If EVEX.vvvv != 1111B.

## VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Singed Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7B /r VCVTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 7B /r VCVTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 7B /r VCVTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}	A	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts eight packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where w represents the number of bits in the destination format) is returned.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VCVTPS2QQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_QuadInteger(SRC[k+31:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTPS2QQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b == 1)

THEN

DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_QuadInteger(SRC[31:0])

ELSE

DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_QuadInteger(SRC[k+31:k])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPS2QQ __m512i _mm512_cvtps_epi64( __m512 a);
VCVTPS2QQ __m512i _mm512_mask_cvtps_epi64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_cvt_roundps_epi64( __m512 a, int r);
VCVTPS2QQ __m512i _mm512_mask_cvt_roundps_epi64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i _mm512_maskz_cvt_roundps_epi64( __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i _mm256_cvtps_epi64( __m256 a);
VCVTPS2QQ __m256i _mm256_mask_cvtps_epi64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i _mm256_maskz_cvtps_epi64( __mmask8 k, __m256 a);
VCVTPS2QQ __m128i _mm_cvtps_epi64( __m128 a);
VCVTPS2QQ __m128i _mm_mask_cvtps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i _mm_maskz_cvtps_epi64( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3

#UD                      If EVEX.vvvv != 1111B.

## VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 79 /r VCVTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from zmm2/m64/m32bcst to two packed unsigned quadword values in zmm1 subject to writemask k1.
EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 79 /r VCVTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}	A	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VCVTPS2UQQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_UQuadInteger(SRC[k+31:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTPS2UQQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b == 1)

THEN

DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_UQuadInteger(SRC[31:0])

ELSE

DEST[i+63:i] ←

Convert\_Single\_Precision\_To\_UQuadInteger(SRC[k+31:k])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTPS2UQQ __m512i __mm512_cvtps_epu64( __m512 a);
VCVTPS2UQQ __m512i __mm512_mask_cvtps_epu64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i __mm512_maskz_cvtps_epu64( __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i __mm512_cvt_roundps_epu64( __m512 a, int r);
VCVTPS2UQQ __m512i __mm512_mask_cvt_roundps_epu64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i __mm512_maskz_cvt_roundps_epu64( __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i __mm256_cvtps_epu64( __m256 a);
VCVTPS2UQQ __m256i __mm256_mask_cvtps_epu64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i __mm256_maskz_cvtps_epu64( __mmask8 k, __m256 a);
VCVTPS2UQQ __m128i __mm_cvtps_epu64( __m128 a);
VCVTPS2UQQ __m128i __mm_mask_cvtps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i __mm_maskz_cvtps_epu64( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3

#UD                      If EVEX.vvvv != 1111B.



## VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 E6 /r VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed quadword integers from xmm2/m128/m64bcst to packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 E6 /r VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 E6 /r VCVTQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

**VCVTQQ2PD (EVEX2 encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_QuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTQQ2PD (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b == 1)

THEN

DEST[i+63:i] ←

Convert\_QuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[63:0])

ELSE

DEST[i+63:i] ←

Convert\_QuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTQQ2PD \_\_m512d \_\_mm512\_cvtepi64\_pd( \_\_m512i a);

VCVTQQ2PD \_\_m512d \_\_mm512\_mask\_cvtepi64\_pd( \_\_m512d s, \_\_mmask16 k, \_\_m512i a);

VCVTQQ2PD \_\_m512d \_\_mm512\_maskz\_cvtepi64\_pd( \_\_mmask16 k, \_\_m512i a);

VCVTQQ2PD \_\_m512d \_\_mm512\_cvt\_roundepi64\_pd( \_\_m512i a, int r);

VCVTQQ2PD \_\_m512d \_\_mm512\_mask\_cvt\_roundepi64\_pd( \_\_m512d s, \_\_mmask8 k, \_\_m512i a, int r);

VCVTQQ2PD \_\_m512d \_\_mm512\_maskz\_cvt\_roundepi64\_pd( \_\_mmask8 k, \_\_m512i a, int r);

VCVTQQ2PD \_\_m256d \_\_mm256\_mask\_cvtepi64\_pd( \_\_m256d s, \_\_mmask8 k, \_\_m256i a);

VCVTQQ2PD \_\_m256d \_\_mm256\_maskz\_cvtepi64\_pd( \_\_mmask8 k, \_\_m256i a);

VCVTQQ2PD \_\_m128d \_\_mm\_mask\_cvtepi64\_pd( \_\_m128d s, \_\_mmask8 k, \_\_m128i a);

VCVTQQ2PD \_\_m128d \_\_mm\_maskz\_cvtepi64\_pd( \_\_mmask8 k, \_\_m128i a);

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

## VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed quadword integers from xmm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed quadword integers from ymm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.0F.W1 5B /r VCVTQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed quadword integers from zmm2/mem to eight packed single-precision floating-point values in ymm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed quadword integers in the source operand (second operand) to packed single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a YMM/XMM/XMM (lower 64 bits) register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

**VCVTQQ2PS (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[k+31:k] ←
      Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[k+31:k] remains unchanged*
    ELSE                           ; zeroing-masking
      DEST[k+31:k] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0

```

**VCVTQ2PS (EVEX encoded versions) when src operand is a memory source**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[k+31:k] ←
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[k+31:k] ←
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[j+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[k+31:k] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[k+31:k] ← 0
        FI
      FI;
    ENDIF;
  DEST[MAXVL-1:VL/2] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTQ2PS __m256 __mm512_cvtepi64_ps( __m512i a);
VCVTQ2PS __m256 __mm512_mask_cvtepi64_ps( __m256 s, __mmask16 k, __m512i a);
VCVTQ2PS __m256 __mm512_maskz_cvtepi64_ps( __mmask16 k, __m512i a);
VCVTQ2PS __m256 __mm512_cvt_roundepi64_ps( __m512i a, int r);
VCVTQ2PS __m256 __mm512_mask_cvt_roundepi64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTQ2PS __m256 __mm512_maskz_cvt_roundepi64_ps( __mmask8 k, __m512i a, int r);
VCVTQ2PS __m128 __mm256_cvtepi64_ps( __m256i a);
VCVTQ2PS __m128 __mm256_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTQ2PS __m128 __mm256_maskz_cvtepi64_ps( __mmask8 k, __m256i a);
VCVTQ2PS __m128 __mm_cvtepi64_ps( __m128i a);
VCVTQ2PS __m128 __mm_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTQ2PS __m128 __mm_maskz_cvtepi64_ps( __mmask8 k, __m128i a);

```

#### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

## VCVTSD2USI—Convert Scalar Double-Precision Floating-Point Value to Unsigned Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.OF.W0 79 /r VCVTSD2USI r32, xmm1/m64{er}	A	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.
EVEX.LIG.F2.OF.W1 79 /r VCVTSD2USI r64, xmm1/m64{er}	A	V/N.E. <sup>1</sup>	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.

### NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

### Operation

#### VCVTSD2USI (EVEX encoded version)

IF (SRC \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode and OperandSize = 64

THEN DEST[63:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[63:0]);

ELSE DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[63:0]);

FI

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2USI unsigned int \_\_mm\_cvtsd\_u32(\_\_m128d);

VCVTSD2USI unsigned int \_\_mm\_cvt\_roundsd\_u32(\_\_m128d, int r);

VCVTSD2USI unsigned \_\_int64 \_\_mm\_cvtsd\_u64(\_\_m128d);

VCVTSD2USI unsigned \_\_int64 \_\_mm\_cvt\_roundsd\_u64(\_\_m128d, int r);

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.0F.W0 79 /r VCVTSS2USI r32, xmm1/m32{er}	A	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.
EVEX.LIG.F3.0F.W1 79 /r VCVTSS2USI r64, xmm1/m32{er}	A	V/N.E. <sup>1</sup>	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.

### NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

#### VCVTSS2USI (EVEX encoded version)

IF (SRC \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[31:0]);

FI;

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned \_\_mm\_cvtss\_u32( \_\_m128 a);

VCVTSS2USI unsigned \_\_mm\_cvt\_roundss\_u32( \_\_m128 a, int r);

VCVTSS2USI unsigned \_\_int64 \_\_mm\_cvtss\_u64( \_\_m128 a);

VCVTSS2USI unsigned \_\_int64 \_\_mm\_cvt\_roundss\_u64( \_\_m128 a, int r);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTTPD2QQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0FW1 7A /r VCVTTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from zmm2/m128/m64bcst to two packed quadword integers in zmm1 using truncation with writemask k1.
EVEX.256.66.0FW1 7A /r VCVTTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0FW1 7A /r VCVTTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/m512 to eight packed quadword integers in zmm1 using truncation with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where w represents the number of bits in the destination format) is returned.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTTPD2QQ (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Double\_Precision\_Floating\_Point\_To\_QuadInteger\_Truncate(SRC[i+63:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0



**VCVTTPD2QQ (EVEX encoded version) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ELSE
      FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPD2QQ __m512i __mm512_cvttpd_epi64( __m512d a);
VCVTTPD2QQ __m512i __mm512_mask_cvttpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i __mm512_maskz_cvttpd_epi64( __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i __mm512_cvtt_roundpd_epi64( __m512d a, int sae);
VCVTTPD2QQ __m512i __mm512_mask_cvtt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m512i __mm512_maskz_cvtt_roundpd_epi64( __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m256i __mm256_mask_cvttpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i __mm256_maskz_cvttpd_epi64( __mmask8 k, __m256d a);
VCVTTPD2QQ __m128i __mm_mask_cvttpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i __mm_maskz_cvttpd_epi64( __mmask8 k, __m128d a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPD2UDQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 78 /r VCVTTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.0F.W1 78 02 /r VCVTTPD2UDQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.0F.W1 78 /r VCVTTPD2UDQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation****VCVTTPD2UDQ (EVEX encoded versions) when src2 operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      DEST[i+31:i] ←
        Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
DEST[MAXVL-1:VL/2] ← 0

```

**VCVTTPD2UDQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    FI;
  ENDFOR
DEST[MAXVL-1:VL/2] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPD2UDQ __m256i __mm512_cvttpd_epu32( __m512d a);
VCVTTPD2UDQ __m256i __mm512_mask_cvttpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i __mm512_maskz_cvttpd_epu32( __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i __mm512_cvtt_roundpd_epu32( __m512d a, int sae);
VCVTTPD2UDQ __m256i __mm512_mask_cvtt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m256i __mm512_maskz_cvtt_roundpd_epu32( __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m128i __mm256_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i __mm256_maskz_cvttpd_epu32( __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i __mm_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __m128i __mm_maskz_cvttpd_epu32( __mmask8 k, __m128d a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD                      If EVEX.vvvv != 1111B.

## VCVTTPD2UQQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 78 /r VCVTTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed unsigned quadword integers in xmm1 using truncation with writemask k1.
EVEX.256.66.0F.W1 78 /r VCVTTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed unsigned quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0F.W1 78 /r VCVTTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTTPD2UQQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger\_Truncate(SRC[i+63:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTTPD2UQQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[j+63:i])
      FI;
    ELSE
      IF *merging-masking*                ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                                ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPD2UQQ __mm<size>[_mask[z]]_cvtt[_round]pd_epu64
VCVTTPD2UQQ __m512i __mm512_cvttpd_epu64( __m512d a);
VCVTTPD2UQQ __m512i __mm512_mask_cvttpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_maskz_cvttpd_epu64( __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_cvtt_roundpd_epu64( __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_mask_cvtt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_maskz_cvtt_roundpd_epu64( __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i __mm256_mask_cvttpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i __mm256_maskz_cvttpd_epu64( __mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i __mm_mask_cvttpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i __mm_maskz_cvttpd_epu64( __mmask8 k, __m128d a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPS2UDQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W0 78 /r VCVTTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.0F.W0 78 /r VCVTTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.0F.W0 78 /r VCVTTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	A	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

**VCVTTPS2UDQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

  IF  $k1[j]$  OR \*no writemask\*

    THEN  $DEST[i+31:i] \leftarrow$

      Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[i+31:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

$DEST[i+31:i] \leftarrow 0$

  FI

  FI;

ENDFOR

$DEST[MAXVL-1:VL] \leftarrow 0$

**VCVTTPS2UDQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPS2UDQ __m512i __mm512_cvttps_epu32( __m512 a);
VCVTTPS2UDQ __m512i __mm512_mask_cvttps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i __mm512_maskz_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i __mm512_cvtt_roundps_epu32( __m512 a, int sae);
VCVTTPS2UDQ __m512i __mm512_mask_cvtt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i __mm512_maskz_cvtt_roundps_epu32( __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i __mm256_mask_cvttps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i __mm256_maskz_cvttps_epu32( __mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i __mm_mask_cvttps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i __mm_maskz_cvttps_epu32( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.



## VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Singed Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7A /r VCVTTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 7A /r VCVTTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 7A /r VCVTTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	A	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ( $2^{w-1}$ , where  $w$  represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

**VCVTTPS2QQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_Single\_Precision\_To\_QuadInteger\_Truncate(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTTPS2QQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ←
            Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
            Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
      FI;
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPS2QQ __m512i __mm512_cvttps_epi64( __m256 a);
VCVTTPS2QQ __m512i __mm512_mask_cvttps_epi64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i __mm512_maskz_cvttps_epi64( __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i __mm512_cvtt_roundps_epi64( __m256 a, int sae);
VCVTTPS2QQ __m512i __mm512_mask_cvtt_roundps_epi64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m512i __mm512_maskz_cvtt_roundps_epi64( __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i __mm256_mask_cvttps_epi64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m256i __mm256_maskz_cvttps_epi64( __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i __mm_mask_cvttps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i __mm_maskz_cvttps_epi64( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.

## VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 78 /r VCVTTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 78 /r VCVTTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 78 /r VCVTTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	A	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

**VCVTTPS2UQQ (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j \* 64

    k ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ←

            Convert\_Single\_Precision\_To\_UQuadInteger\_Truncate(SRC[k+31:k])

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE ; zeroing-masking

            DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTTPS2UQQ (EVEX encoded versions) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF;
  ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTTPS2UQQ __mm<size>[_mask[z]]_cvtt[_round]ps_epu64
VCVTTPS2UQQ __m512i __mm512_cvttps_epu64( __m256 a);
VCVTTPS2UQQ __m512i __mm512_mask_cvttps_epu64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_maskz_cvttps_epu64( __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_cvtt_roundps_epu64( __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_mask_cvtt_roundps_epu64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_maskz_cvtt_roundps_epu64( __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m256i __mm256_mask_cvttps_epu64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m256i __mm256_maskz_cvttps_epu64( __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i __mm_mask_cvttps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i __mm_maskz_cvttps_epu64( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.

## VCVTTSD2USI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.OF.W0 78 /r VCVTTSD2USI r32, xmm1/m64{sae}	A	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.
EVEX.LIG.F2.OF.W1 78 /r VCVTTSD2USI r64, xmm1/m64{sae}	A	V/N.E. <sup>1</sup>	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

### Operation

#### VCVTTSD2USI (EVEX encoded version)

IF 64-Bit Mode and OperandSize = 64

```
THEN  DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
ELSE  DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
```

FI

### Intel C/C++ Compiler Intrinsic Equivalent

```
VCVTTSD2USI unsigned int _mm_cvtsd_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvtsd_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundsd_u64(__m128d, int sae);
```

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.0F.W0 78 /r VCVTTSS2USI r32, xmm1/m32{sae}	A	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.
EVEX.LIG.F3.0F.W1 78 /r VCVTTSS2USI r64, xmm1/m32{sae}	A	V/N.E. <sup>1</sup>	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts with truncation a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value  $2^w - 1$  is returned, where  $w$  represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation****VCVTSS2USI (EVEX encoded version)**

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[31:0]);

FI;

**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTSS2USI unsigned int \_mm\_cvttss\_u32( \_\_m128 a);

VCVTSS2USI unsigned int \_mm\_cvtt\_roundss\_u32( \_\_m128 a, int sae);

VCVTSS2USI unsigned \_\_int64 \_mm\_cvttss\_u64( \_\_m128 a);

VCVTSS2USI unsigned \_\_int64 \_mm\_cvtt\_roundss\_u64( \_\_m128 a, int sae);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W0 7A /r VCVTUDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512F	Convert two packed unsigned doubleword integers from ymm2/m64/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.
EVEX.256.F3.0F.W0 7A /r VCVTUDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.
EVEX.512.F3.0F.W0 7A /r VCVTUDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512F	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed unsigned doubleword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Attempt to encode this instruction with EVEX embedded rounding is ignored.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTUDQ2PD (EVEX encoded versions) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

      Convert\_UInteger\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0



**VCVTUDQ2PD (EVEX encoded versions) when src operand is a memory source**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            Convert_UIInteger_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+63:i] ←
            Convert_UIInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
  
```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTUDQ2PD __m512d __mm512_cvtepu32_pd( __m256i a);
VCVTUDQ2PD __m512d __mm512_mask_cvtepu32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d __mm512_maskz_cvtepu32_pd( __mmask8 k, __m256i a);
VCVTUDQ2PD __m256d __mm256_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m256d __mm256_mask_cvtepu32_pd( __m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d __mm256_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d __mm_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m128d __mm_mask_cvtepu32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d __mm_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
  
```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E5.

#UD If EVEX.vvvv != 1111B.

## VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W0 7A /r VCVTUDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F2.0F.W0 7A /r VCVTUDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F2.0F.W0 7A /r VCVTUDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	A	V/V	AVX512F	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed unsigned doubleword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTUDQ2PS (EVEX encoded version) when src operand is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_UIInteger\_To\_Single\_Precision\_Floating\_Point(SRC[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTUDQ2PS (EVEX encoded version) when src operand is a memory source**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            Convert_UIInteger_To_Single_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
            Convert_UIInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTUDQ2PS __m512 __mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m512 __mm512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_cvt_roundedu32_ps( __m512i a, int r);
VCVTUDQ2PS __m512 __mm512_mask_cvt_roundedu32_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 __mm512_maskz_cvt_roundedu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 __mm256_cvtepu32_ps( __m256i a);
VCVTUDQ2PS __m256 __mm256_mask_cvtepu32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 __mm256_maskz_cvtepu32_ps( __mmask8 k, __m256i a);
VCVTUDQ2PS __m128 __mm_cvtepu32_ps( __m128i a);
VCVTUDQ2PS __m128 __mm_mask_cvtepu32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 __mm_maskz_cvtepu32_ps( __mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 7A /r VCVTUQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 7A /r VCVTUQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 7A /r VCVTUQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed unsigned quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTUQQ2PD (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL == 512) AND (EVEX.b == 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

Convert\_UQuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VCVTUQQ2PD (EVEX encoded version) when src operand is a memory source**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ←
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+63:i] ←
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VCVTUQQ2PD __m512d __mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d __mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d __mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d __mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d __mm512_mask_cvt_roundepu64_ps( __m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d __mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d __mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d __mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d __mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m128d __mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PD __m128d __mm_mask_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d __mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single-precision floating-point values in zmm1 with writemask k1.
EVEX.256.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.F2.0F.W1 7A /r VCVTUQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single-precision floating-point values in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts packed unsigned quadword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

**VCVTUQQ2PS (EVEX encoded version) when src operand is a register**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

Convert\_UQuadInteger\_To\_Single\_Precision\_Floating\_Point(SRC[k+63:k])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0

**VCVTUQQ2PS (EVEX encoded version) when src operand is a memory source**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] ←
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking*                ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                                  ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VCVTUQQ2PS __m256 __mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PS __m256 __mm512_mask_cvtepu64_ps( __m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __mm512_cvt_roundedu64_ps( __m512i a, int r);
VCVTUQQ2PS __m256 __mm512_mask_cvt_roundedu64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 __mm512_maskz_cvt_roundedu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128 __mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PS __m128 __mm256_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 __mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 __mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PS __m128 __mm_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 __mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

```

#### SIMD Floating-Point Exceptions

Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

**VCVTUSI2SD—Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.F2.0F.W0 7B /r VCVTUSI2SD xmm1, xmm2, r/m32	A	V/V	AVX512F	Convert one unsigned doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W1 7B /r VCVTUSI2SD xmm1, xmm2, r/m64{er}	A	V/N.E. <sup>1</sup>	AVX512F	Convert one unsigned quadword integer from r/m64 to one double-precision floating-point value in xmm1.

**NOTES:**

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

**Description**

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

**Operation****VCVTUSI2SD (EVEX encoded version)**

IF (SRC2 \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_UInteger\_To\_Double\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

DEST[63:0] ← Convert\_UInteger\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAXVL-1:128] ← 0



**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTUSI2SD \_\_m128d \_mm\_cvtsi2sdq\_epi64(\_\_m128d s, unsigned a);  
VCVTUSI2SD \_\_m128d \_mm\_cvtsi2sdq\_epi64(\_\_m128d s, unsigned \_\_int64 a);  
VCVTUSI2SD \_\_m128d \_mm\_cvtsi2sdq\_epi64(\_\_m128d s, unsigned \_\_int64 a, int r);

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

See Exceptions Type E3NF if W1, else type E10NF.

## VCVTUSI2SS—Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.F3.0F.W0 7B /r VCVTUSI2SS xmm1, xmm2, r/m32{er}	A	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W1 7B /r VCVTUSI2SS xmm1, xmm2, r/m64{er}	A	V/N.E. <sup>1</sup>	AVX512F	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.

### NOTES:

- For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Converts a unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

### Operation

#### VCVTUSI2SS (EVEX encoded version)

IF (SRC2 \*is register\*) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAXVL-1:128] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SS \_\_m128 \_\_mm\_cvts32\_ss( \_\_m128 s, unsigned a);

VCVTUSI2SS \_\_m128 \_\_mm\_cvt\_rounds32\_ss( \_\_m128 s, unsigned a, int r);

VCVTUSI2SS \_\_m128 \_\_mm\_cvts64\_ss( \_\_m128 s, unsigned \_\_int64 a);

VCVTUSI2SS \_\_m128 \_\_mm\_cvt\_rounds64\_ss( \_\_m128 s, unsigned \_\_int64 a, int r);

**SIMD Floating-Point Exceptions**

Precision

**Other Exceptions**

See Exceptions Type E3NF.

**VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 42 /r ib VDBPSADBW xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL AVX512BW	Compute packed SAD word results of unsigned bytes in dword block from xmm2 with unsigned bytes of dword blocks transformed from xmm3/m128 using the shuffle controls in imm8. Results are written to xmm1 under the writemask k1.
EVEX.NDS.256.66.0F3A.W0 42 /r ib VDBPSADBW ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL AVX512BW	Compute packed SAD word results of unsigned bytes in dword block from ymm2 with unsigned bytes of dword blocks transformed from ymm3/m256 using the shuffle controls in imm8. Results are written to ymm1 under the writemask k1.
EVEX.NDS.512.66.0F3A.W0 42 /r ib VDBPSADBW zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW	Compute packed SAD word results of unsigned bytes in dword block from zmm2 with unsigned bytes of dword blocks transformed from zmm3/m512 using the shuffle controls in imm8. Results are written to zmm1 under the writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

**Description**

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each super block of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see “Tmp1” in Figure 5-8, is constructed from the second source operand the imm8 byte as shuffle control to select dword elements within a 128-bit lane of the second source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 0 and 1 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 0.
- The next two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from the intermediate vector Tmp1 with a second stationary dword element in the corresponding qword superblock of the first source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 2 and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.
- The intermediate vector is constructed in 128-bits lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the intermediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.

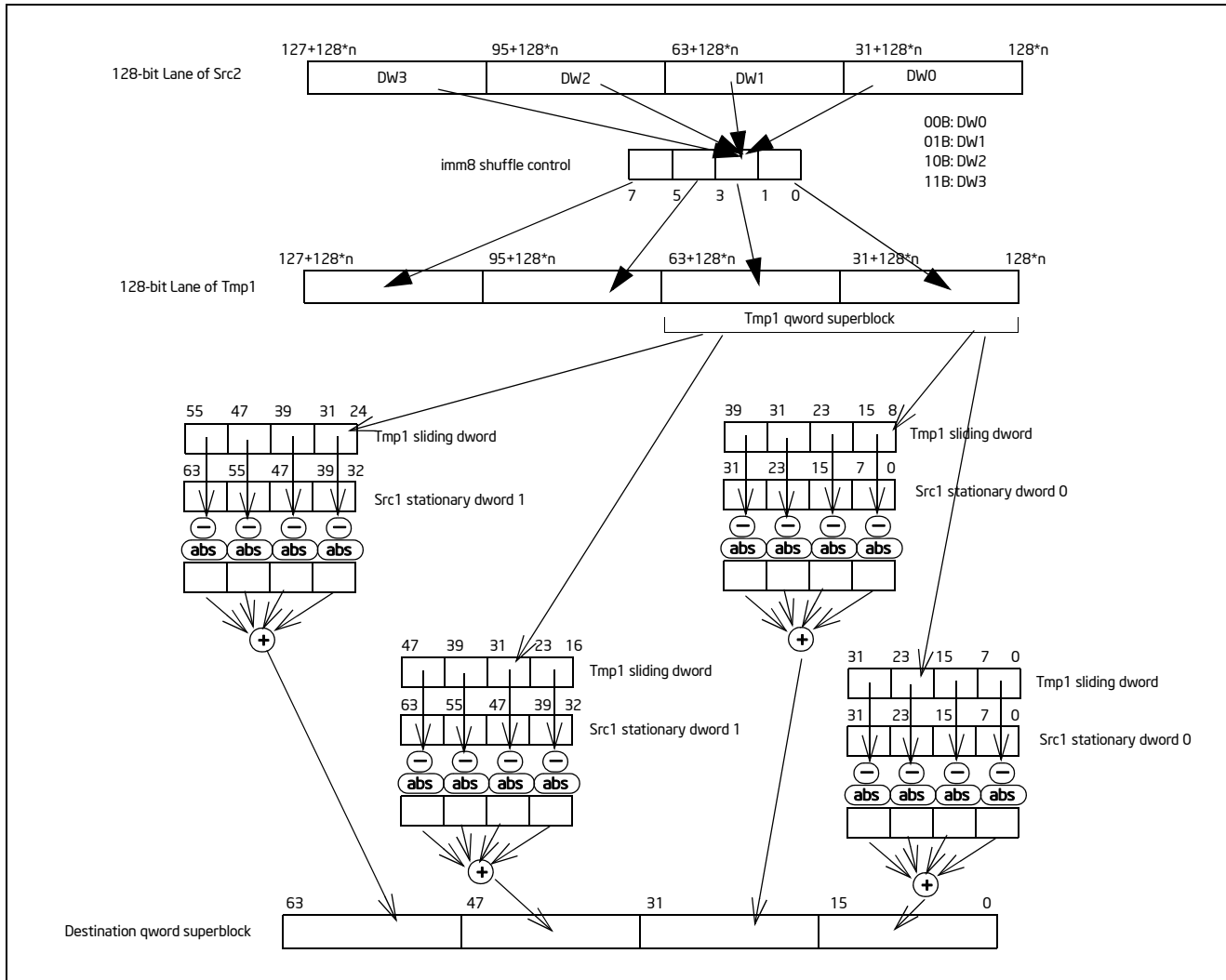


Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBW

**Operation****VDBPSADBW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

Selection of quadruplets:

FOR I = 0 to VL step 128

TMP1[I+31:I] ← select (SRC2[I+127:I], imm8[1:0])

TMP1[I+63:I+32] ← select (SRC2[I+127:I], imm8[3:2])

TMP1[I+95:I+64] ← select (SRC2[I+127:I], imm8[5:4])

TMP1[I+127:I+96] ← select (SRC2[I+127:I], imm8[7:6])

END FOR

SAD of quadruplets:

FOR I = 0 to VL step 64

    TMP\_DEST[I+15:I] ← ABS(SRC1[I+7:I] - TMP1[I+7:I]) +  
         ABS(SRC1[I+15:I+8] - TMP1[I+15:I+8]) +  
         ABS(SRC1[I+23:I+16] - TMP1[I+23:I+16]) +  
         ABS(SRC1[I+31:I+24] - TMP1[I+31:I+24])

    TMP\_DEST[I+31:I+16] ← ABS(SRC1[I+7:I] - TMP1[I+15:I+8]) +  
         ABS(SRC1[I+15:I+8] - TMP1[I+23:I+16]) +  
         ABS(SRC1[I+23:I+16] - TMP1[I+31:I+24]) +  
         ABS(SRC1[I+31:I+24] - TMP1[I+39:I+32])

    TMP\_DEST[I+47:I+32] ← ABS(SRC1[I+39:I+32] - TMP1[I+23:I+16]) +  
         ABS(SRC1[I+47:I+40] - TMP1[I+31:I+24]) +  
         ABS(SRC1[I+55:I+48] - TMP1[I+39:I+32]) +  
         ABS(SRC1[I+63:I+56] - TMP1[I+47:I+40])

    TMP\_DEST[I+63:I+48] ← ABS(SRC1[I+39:I+32] - TMP1[I+31:I+24]) +  
         ABS(SRC1[I+47:I+40] - TMP1[I+39:I+32]) +  
         ABS(SRC1[I+55:I+48] - TMP1[I+47:I+40]) +  
         ABS(SRC1[I+63:I+56] - TMP1[I+55:I+48])

ENDFOR

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← TMP\_DEST[i+15:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m128i _mm_dbsad_epu8(__m128i a, __m128i b);
VDBPSADBW __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b);
VDBPSADBW __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4NF.nb.

## VEXPANDPD—Load Sparse Packed Double-Precision Floating-Point Values from Dense Memory

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 88 /r VEXPANDPD xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 88 /r VEXPANDPD ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 88 /r VEXPANDPD zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F	Expand packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Expand (load) up to 8/4/2, contiguous, double-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VEXPANDPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

$k \leftarrow 0$

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

IF  $k1[j]$  OR \*no writemask\*

THEN

$DEST[i+63:i] \leftarrow SRC[k+63:k];$

$k \leftarrow k + 64$

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

THEN  $DEST[i+63:i] \leftarrow 0$

FI

FI;

ENDFOR

$DEST[MAXVL-1:VL] \leftarrow 0$



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDPD __m512d __mm512_mask_expand_pd( __m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d __mm512_maskz_expand_pd( __mmask8 k, __m512d a);
VEXPANDPD __m512d __mm512_mask_expandloadu_pd( __m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d __mm512_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m256d __mm256_mask_expand_pd( __m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d __mm256_maskz_expand_pd( __mmask8 k, __m256d a);
VEXPANDPD __m256d __mm256_mask_expandloadu_pd( __m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d __mm256_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m128d __mm_mask_expand_pd( __m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d __mm_maskz_expand_pd( __mmask8 k, __m128d a);
VEXPANDPD __m128d __mm_mask_expandloadu_pd( __m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d __mm_maskz_expandloadu_pd( __mmask8 k, void * a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.nb.

#UD                      If EVEX.vvvv != 1111B.

## VEXPANDPS—Load Sparse Packed Single-Precision Floating-Point Values from Dense Memory

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 88 /r VEXPANDPS xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F	Expand packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Expand (load) up to 16/8/4, contiguous, single-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VEXPANDPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

$k \leftarrow 0$

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

IF  $k1[j]$  OR \*no writemask\*

THEN

$DEST[i+31:i] \leftarrow SRC[k+31:k];$

$k \leftarrow k + 32$

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+31:i] remains unchanged\*

ELSE ; zeroing-masking

$DEST[i+31:i] \leftarrow 0$

FI

FI;

ENDFOR

$DEST[MAXVL-1:VL] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDPS __m512 __mm512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 __mm512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 __mm512_mask_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 __mm512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 __mm256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 __mm256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 __mm256_mask_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 __mm256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 __mm_mask_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 __mm_maskz_expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 __mm_mask_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 __mm_maskz_expandloadu_ps( __mmask8 k, void * a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.nb.

#UD                      If EVEX.vvvv != 1111B.

**VERR/VERW—Verify a Segment for Reading or Writing**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /4	VERR <i>r/m16</i>	M	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be read.
OF 00 /5	VERW <i>r/m16</i>	M	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be written.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m ( <i>r</i> )	NA	NA	NA

**Description**

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not NULL.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment's DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

**Operation**

```
IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))
  THEN ZF ← 0; FI;
```

Read segment descriptor;

```
IF SegmentDescriptor(DescriptorType) = 0 (* System segment *)
or (SegmentDescriptor(Type) ≠ conforming code segment)
and (CPL > DPL) or (RPL > DPL)
```

```
  THEN
```

```
    ZF ← 0;
```

```
  ELSE
```

```
    IF ((Instruction = VERR) and (Segment readable))
```

```
    or ((Instruction = VERW) and (Segment writable))
```

```
      THEN
```

```
        ZF ← 1;
```

```
    FI;
```

```
FI;
```

## Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

## Protected Mode Exceptions

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## Real-Address Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in real-address mode. If the LOCK prefix is used.
-----	--

## Virtual-8086 Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in virtual-8086 mode. If the LOCK prefix is used.
-----	--

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEXTRACTF128 xmm1/m128, ymm2, imm8	A	V/V	AVX	Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.
EVEX.256.66.0F3A.W0 19 /r ib VEXTRACTF32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	AVX512VL AVX512F	Extract 128 bits of packed single-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 19 /r ib VEXTRACTF32x4 xmm1/m128 {k1}{z}, zmm2, imm8	C	V/V	AVX512F	Extract 128 bits of packed single-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.256.66.0F3A.W1 19 /r ib VEXTRACTF64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	AVX512VL AVX512DQ	Extract 128 bits of packed double-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 19 /r ib VEXTRACTF64X2 xmm1/m128 {k1}{z}, zmm2, imm8	B	V/V	AVX512DQ	Extract 128 bits of packed double-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 1B /r ib VEXTRACTF32X8 ymm1/m256 {k1}{z}, zmm2, imm8	D	V/V	AVX512DQ	Extract 256 bits of packed single-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEX.512.66.0F3A.W1 1B /r ib VEXTRACTF64x4 ymm1/m256 {k1}{z}, zmm2, imm8	C	V/V	AVX512F	Extract 256 bits of packed double-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

VEXTRACTF128/VEXTRACTF32x4 and VEXTRACTF64x2 extract 128-bits of single-precision floating-point values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTF32x8 and VEXTRACTF64x4 extract 256-bits of double-precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 6 bits of the immediate are ignored.

If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

**Operation****VEEXTRACTF32x4 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC1[127:0]

1: TMP\_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC1[127:0]

01: TMP\_DEST[127:0] ← SRC1[255:128]

10: TMP\_DEST[127:0] ← SRC1[383:256]

11: TMP\_DEST[127:0] ← SRC1[511:384]

ESAC.

FI;

FOR j ← 0 TO 3

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:128] ← 0

**VEEXTRACTF32x4 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC1[127:0]

1: TMP\_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC1[127:0]

01: TMP\_DEST[127:0] ← SRC1[255:128]

10: TMP\_DEST[127:0] ← SRC1[383:256]

11: TMP\_DEST[127:0] ← SRC1[511:384]

ESAC.

FI;

FOR j ← 0 TO 3

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE \*DEST[i+31:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

**VEXTRACTF64x2 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
```

FI;

IF VL = 512

```
CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.
```

FI;

FOR j ← 0 TO 1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:128] ← 0

**VEXTRACTF64x2 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

```
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  1: TMP_DEST[127:0] ← SRC1[255:128]
ESAC.
```

FI;

IF VL = 512

```
CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] ← SRC1[127:0]
  01: TMP_DEST[127:0] ← SRC1[255:128]
  10: TMP_DEST[127:0] ← SRC1[383:256]
  11: TMP_DEST[127:0] ← SRC1[511:384]
ESAC.
```

FI;

FOR j ← 0 TO 1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]



```

        ELSE *DEST[i+63:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR

```

#### **VEEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is a register**

```

VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*      ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*    ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:256] ← 0

```

#### **VEEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is memory**

```

CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR

```

#### **VEEXTRACTF64x4 (EVEX.512 encoded version) when destination is a register**

```

VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*      ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*    ; zeroing-masking
        FI
    FI;
ENDFOR

```

```

                DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:256] ← 0

```

**VEXTRACTF64x4 (EVEX.512 encoded version) when destination is memory**

```

CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        ; merging-masking
        *DEST[i+63:i] remains unchanged*
    FI;
ENDFOR

```

**VEXTRACTF128 (memory destination form)**

```

CASE (imm8[0]) OF
    0: DEST[127:0] ← SRC1[127:0]
    1: DEST[127:0] ← SRC1[255:128]
ESAC.

```

**VEXTRACTF128 (register destination form)**

```

CASE (imm8[0]) OF
    0: DEST[127:0] ← SRC1[127:0]
    1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXTRACTF32x4 __m128_mm512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128_mm512_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128_mm512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128_mm256_extractf32x4_ps(__m256 a, const int nidx);
VEXTRACTF32x4 __m128_mm256_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x4 __m128_mm256_maskz_extractf32x4_ps(__mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x8 __m256_mm512_extractf32x8_ps(__m512 a, const int nidx);
VEXTRACTF32x8 __m256_mm512_mask_extractf32x8_ps(__m256 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x8 __m256_mm512_maskz_extractf32x8_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF64x2 __m128d_mm512_extractf64x2_pd(__m512d a, const int nidx);
VEXTRACTF64x2 __m128d_mm512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d_mm512_maskz_extractf64x2_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d_mm256_extractf64x2_pd(__m256d a, const int nidx);
VEXTRACTF64x2 __m128d_mm256_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x2 __m128d_mm256_maskz_extractf64x2_pd(__mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x4 __m256d_mm512_extractf64x4_pd(__m512d a, const int nidx);
VEXTRACTF64x4 __m256d_mm512_mask_extractf64x4_pd(__m256d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d_mm512_maskz_extractf64x4_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF128 __m128_mm256_extractf128_ps(__m256 a, int offset);

```

VEXTRACTF128 \_\_m128d \_mm256\_extractf128\_pd (\_\_m256d a, int offset);  
 VEXTRACTF128 \_\_m128i \_mm256\_extractf128\_si256(\_\_m256i a, int offset);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 39 /r ib VEXTRACTI128 xmm1/m128, ymm2, imm8	A	V/V	AVX2	Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.
EVEX.256.66.0F3A.W0 39 /r ib VEXTRACTI32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	AVX512VL AVX512F	Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 39 /r ib VEXTRACTI32x4 xmm1/m128 {k1}{z}, zmm2, imm8	C	V/V	AVX512F	Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.256.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	AVX512VL AVX512DQ	Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, zmm2, imm8	B	V/V	AVX512DQ	Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 3B /r ib VEXTRACTI32X8 ymm1/m256 {k1}{z}, zmm2, imm8	D	V/V	AVX512DQ	Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEX.512.66.0F3A.W1 3B /r ib VEXTRACTI64x4 ymm1/m256 {k1}{z}, zmm2, imm8	C	V/V	AVX512F	Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEXTRACTI32x8: The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.

**VEXTRACTI64x4:** The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.

If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### **VEXTRACTI32x4 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC1[127:0]

1: TMP\_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC1[127:0]

01: TMP\_DEST[127:0] ← SRC1[255:128]

10: TMP\_DEST[127:0] ← SRC1[383:256]

11: TMP\_DEST[127:0] ← SRC1[511:384]

ESAC.

FI;

FOR j ← 0 TO 3

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:128] ← 0

#### **VEXTRACTI32x4 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC1[127:0]

1: TMP\_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC1[127:0]

01: TMP\_DEST[127:0] ← SRC1[255:128]

10: TMP\_DEST[127:0] ← SRC1[383:256]

11: TMP\_DEST[127:0] ← SRC1[511:384]

ESAC.

```

FI;

FOR j ← 0 TO 3
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VEXTRACTI64x2 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```

  CASE (imm8[0]) OF
    0: TMP_DEST[127:0] ← SRC1[127:0]
    1: TMP_DEST[127:0] ← SRC1[255:128]
  ESAC.

```

FI;

IF VL = 512

```

  CASE (imm8[1:0]) OF
    00: TMP_DEST[127:0] ← SRC1[127:0]
    01: TMP_DEST[127:0] ← SRC1[255:128]
    10: TMP_DEST[127:0] ← SRC1[383:256]
    11: TMP_DEST[127:0] ← SRC1[511:384]
  ESAC.

```

FI;

```

FOR j ← 0 TO 1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking*      ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking*    ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:128] ← 0

```

**VEXTRACTI64x2 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0]  $\leftarrow$  SRC1[127:0]1: TMP\_DEST[127:0]  $\leftarrow$  SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0]  $\leftarrow$  SRC1[127:0]01: TMP\_DEST[127:0]  $\leftarrow$  SRC1[255:128]10: TMP\_DEST[127:0]  $\leftarrow$  SRC1[383:256]11: TMP\_DEST[127:0]  $\leftarrow$  SRC1[511:384]

ESAC.

FI;

FOR j  $\leftarrow$  0 TO 1i  $\leftarrow$  j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i]  $\leftarrow$  TMP\_DEST[i+63:i]

ELSE \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

**VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is a register**

VL = 512

CASE (imm8[0]) OF

0: TMP\_DEST[255:0]  $\leftarrow$  SRC1[255:0]1: TMP\_DEST[255:0]  $\leftarrow$  SRC1[511:256]

ESAC.

FOR j  $\leftarrow$  0 TO 7i  $\leftarrow$  j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i]  $\leftarrow$  TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i]  $\leftarrow$  0

FI

FI;

ENDFOR

DEST[MAXVL-1:256]  $\leftarrow$  0

**VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory**

CASE (imm8[0]) OF

0: TMP\_DEST[255:0] ← SRC1[255:0]

1: TMP\_DEST[255:0] ← SRC1[511:256]

ESAC.

FOR j ← 0 TO 7

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE \*DEST[i+31:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

**VEXTRACTI64x4 (EVEX.512 encoded version) when destination is a register**

VL = 512

CASE (imm8[0]) OF

0: TMP\_DEST[255:0] ← SRC1[255:0]

1: TMP\_DEST[255:0] ← SRC1[511:256]

ESAC.

FOR j ← 0 TO 3

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:256] ← 0

**VEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory**

CASE (imm8[0]) OF

0: TMP\_DEST[255:0] ← SRC1[255:0]

1: TMP\_DEST[255:0] ← SRC1[511:256]

ESAC.

FOR j ← 0 TO 3

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE \*DEST[i+63:i] remains unchanged\* ; merging-masking

FI;

ENDFOR



**VEXTRACTI128 (memory destination form)**

CASE (imm8[0]) OF

0: DEST[127:0] ← SRC1[127:0]

1: DEST[127:0] ← SRC1[255:128]

ESAC.

**VEXTRACTI128 (register destination form)**

CASE (imm8[0]) OF

0: DEST[127:0] ← SRC1[127:0]

1: DEST[127:0] ← SRC1[255:128]

ESAC.

DEST[MAXVL-1:128] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXTRACTI32x4 __m128i_mm512_extracti32x4_epi32(__m512i a, const int nidx);
VEXTRACTI32x4 __m128i_mm512_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i_mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i_mm256_extracti32x4_epi32(__m256i a, const int nidx);
VEXTRACTI32x4 __m128i_mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x4 __m128i_mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x8 __m256i_mm512_extracti32x8_epi32(__m512i a, const int nidx);
VEXTRACTI32x8 __m256i_mm512_mask_extracti32x8_epi32(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x8 __m256i_mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i_mm512_extracti64x2_epi64(__m512i a, const int nidx);
VEXTRACTI64x2 __m128i_mm512_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i_mm512_maskz_extracti64x2_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i_mm256_extracti64x2_epi64(__m256i a, const int nidx);
VEXTRACTI64x2 __m128i_mm256_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x2 __m128i_mm256_maskz_extracti64x2_epi64(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x4 __m256i_mm512_extracti64x4_epi64(__m512i a, const int nidx);
VEXTRACTI64x4 __m256i_mm512_mask_extracti64x4_epi64(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x4 __m256i_mm512_maskz_extracti64x4_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI128 __m128i_mm256_extracti128_si256(__m256i a, int offset);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VFIXUPIMMPD—Fix Up Special Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 54 /r ib VFIXUPIMMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Fix up special numbers in float64 vector xmm1, float64 vector xmm2 and int64 vector xmm3/m128/m64bcst and store the result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W1 54 /r ib VFIXUPIMMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Fix up special numbers in float64 vector ymm1, float64 vector ymm2 and int64 vector ymm3/m256/m64bcst and store the result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W1 54 /r ib VFIXUPIMMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	A	V/V	AVX512F	Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Perform fix-up of quad-word elements encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.

**Operation**

```
enum TOKEN_TYPE
```

```
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```

```
FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] ← ((src1[62:52] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j ← 7;
    } ; end source special CASE(tsrc...)
```

; The required response from src3 table is extracted  
 token\_response[3:0] = tbl3[3+4\*j:4\*j];

```
CASE(token_response[3:0]) {
    0000: dest[63:0] ← dest[63:0]; ; preserve content of DEST
    0001: dest[63:0] ← tsrc[63:0]; ; pass through src1 normal input value, denormal as zero
    0010: dest[63:0] ← QNaN(tsrc[63:0]);
    0011: dest[63:0] ← QNAN_Indefinite;
    0100: dest[63:0] ← -INF;
    0101: dest[63:0] ← +INF;
    0110: dest[63:0] ← tsrc.sign? -INF : +INF;
    0111: dest[63:0] ← -0;
    1000: dest[63:0] ← +0;
    1001: dest[63:0] ← -1;
    1010: dest[63:0] ← +1;
    1011: dest[63:0] ← ½;
    1100: dest[63:0] ← 90.0;
    1101: dest[63:0] ← PI/2;
    1110: dest[63:0] ← MAX_FLOAT;
    1111: dest[63:0] ← -MAX_FLOAT;
} ; end of token_response CASE
```

```

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[63:0];
} ; end of FIXUPIMM_DP()

```

**VFIXUPIMMPD**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[63:0], imm8 [7:0])
        ELSE
          DEST[i+63:i] ← FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[i+63:i], imm8 [7:0])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE DEST[i+63:i] ← 0 ; zeroing-masking
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

Immediate Control Description:

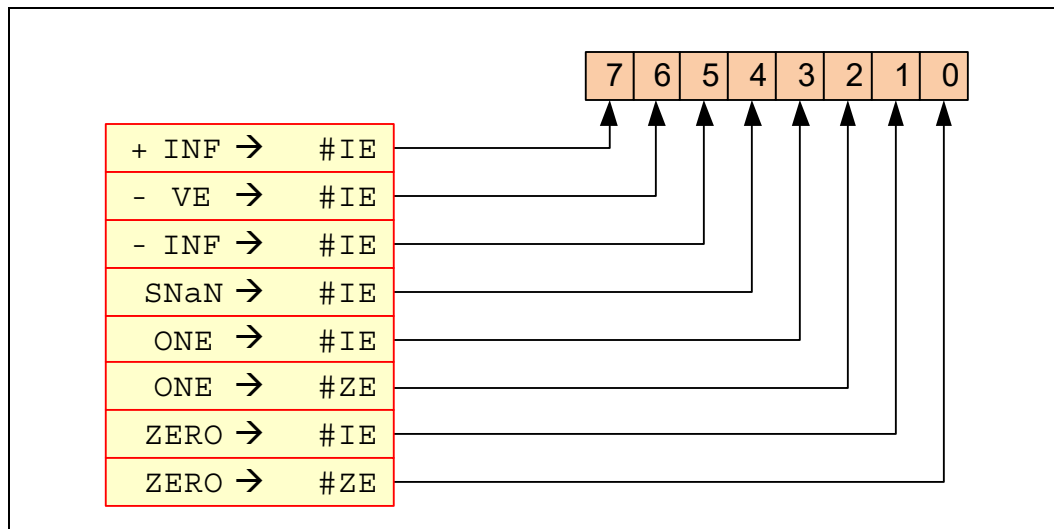


Figure 5-9. VFIXUPIMMPD Immediate Control Description

#### Intel C/C++ Compiler Intrinsic Equivalent

```
VFIXUPIMMPD __m512d __mm512_fixupimm_pd( __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d __mm512_mask_fixupimm_pd( __m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d __mm512_maskz_fixupimm_pd( __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d __mm512_fixupimm_round_pd( __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d __mm512_mask_fixupimm_round_pd( __m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d __mm512_maskz_fixupimm_round_pd( __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m256d __mm256_fixupimm_pd( __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m256d __mm256_mask_fixupimm_pd( __m256d s, __mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m256d __mm256_maskz_fixupimm_pd( __mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m128d __mm128_fixupimm_pd( __m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d __mm128_mask_fixupimm_pd( __m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d __mm128_maskz_fixupimm_pd( __mmask8 k, __m128d a, __m128i tbl, int imm);
```

#### SIMD Floating-Point Exceptions

Zero, Invalid

#### Other Exceptions

See Exceptions Type E2.

## VFIXUPIMMPS—Fix Up Special Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 54 /r VFIXUPIMMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W0 54 /r VFIXUPIMMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W0 54 /r ib VFIXUPIMMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	A	V/V	AVX512F	Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Perform fix-up of doubleword elements encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , yields an incorrect result. To deal with this, VFIXUPIMMPS can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

**Operation**

```

enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}

FIXUPIMM_SP ( dest[31:0], src1[31:0], tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j ← 7;
    } ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] ← dest[31:0]; ; preserve content of DEST
        0001: dest[31:0] ← tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] ← QNaN(tsrc[31:0]);
        0011: dest[31:0] ← QNAN_Indefinite;
        0100: dest[31:0] ← -INF;
        0101: dest[31:0] ← +INF;
        0110: dest[31:0] ← tsrc.sign? -INF : +INF;
        0111: dest[31:0] ← -0;
        1000: dest[31:0] ← +0;
        1001: dest[31:0] ← -1;
        1010: dest[31:0] ← +1;
        1011: dest[31:0] ← ½;
        1100: dest[31:0] ← 90.0;
        1101: dest[31:0] ← PI/2;
        1110: dest[31:0] ← MAX_FLOAT;
        1111: dest[31:0] ← -MAX_FLOAT;
    } ; end of token_response CASE

```

```

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[31:0];
} ; end of FIXUPIMM_SP()

```

**VFIXUPIMMPS (EVEX)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[31:0], imm8 [7:0])
        ELSE
          DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[i+31:i], imm8 [7:0])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE DEST[i+31:i] ← 0 ; zeroing-masking
        FI
      FI;
    ENDIF
  DEST[VL-1:VL] ← 0
ENDFOR

```



## Immediate Control Description:

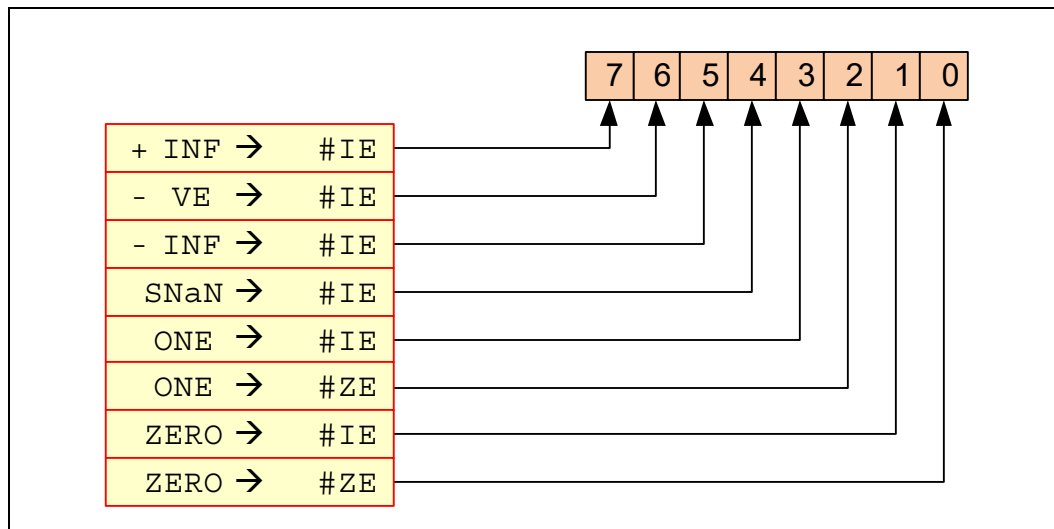


Figure 5-10. VFIXUPIMMPS Immediate Control Description

## Intel C/C++ Compiler Intrinsic Equivalent

```

VFIXUPIMMPS __m512 __mm512_fixupimm_ps( __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_ps( __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 __mm512_fixupimm_round_ps( __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 __mm512_maskz_fixupimm_round_ps( __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m256 __mm256_fixupimm_ps( __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_mask_fixupimm_ps(__m256 s, __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_maskz_fixupimm_ps( __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m128 __mm_fixupimm_ps( __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm_mask_fixupimm_ps(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm_maskz_fixupimm_ps( __mmask8 k, __m128 a, __m128i tbl, int imm);

```

## SIMD Floating-Point Exceptions

Zero, Invalid

## Other Exceptions

See Exceptions Type E2.

## VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 55 /r ib VFIXUPIMMSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F	Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Perform a fix-up of the low quadword element encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , yields an incorrect result. To deal with this, `VFIXUPIMMSD` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

`MXCSR.DAZ` is used and refer to `zmm2` only (i.e. `zmm1` is not considered as zero in case `MXCSR.DAZ` is set).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

### Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```

```

FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] ← ((src1[62:52] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j ← 7;
    } ; end source special CASE(tsrc...)

; The required response from src3 table is extracted
token_response[3:0] = tbl3[3+4*j:4*j];

CASE(token_response[3:0]) {
    0000: dest[63:0] ← dest[63:0] ; preserve content of DEST
    0001: dest[63:0] ← tsrc[63:0]; ; pass through src1 normal input value, denormal as zero
    0010: dest[63:0] ← QNaN(tsrc[63:0]);
    0011: dest[63:0] ← QNaN_Indefinite;
    0100: dest[63:0] ← -INF;
    0101: dest[63:0] ← +INF;
    0110: dest[63:0] ← tsrc.sign? -INF : +INF;
    0111: dest[63:0] ← -0;
    1000: dest[63:0] ← +0;
    1001: dest[63:0] ← -1;
    1010: dest[63:0] ← +1;
    1011: dest[63:0] ← ½;
    1100: dest[63:0] ← 90.0;
    1101: dest[63:0] ← PI/2;
    1110: dest[63:0] ← MAX_FLOAT;
    1111: dest[63:0] ← -MAX_FLOAT;
} ; end of token_response CASE

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[63:0];
} ; end of FIXUPIMM_DP()

```

**VFIXUPIMMSD (EVEX encoded version)**

```
IF k1[0] OR *no writemask*  
  THEN DEST[63:0] ← FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])  
ELSE  
  IF *merging-masking* ; merging-masking  
    THEN *DEST[63:0] remains unchanged*  
    ELSE DEST[63:0] ← 0 ; zeroing-masking  
FI  
FI;  
DEST[127:64] ← SRC1[127:64]  
DEST[MAXVL-1:128] ← 0
```

Immediate Control Description:

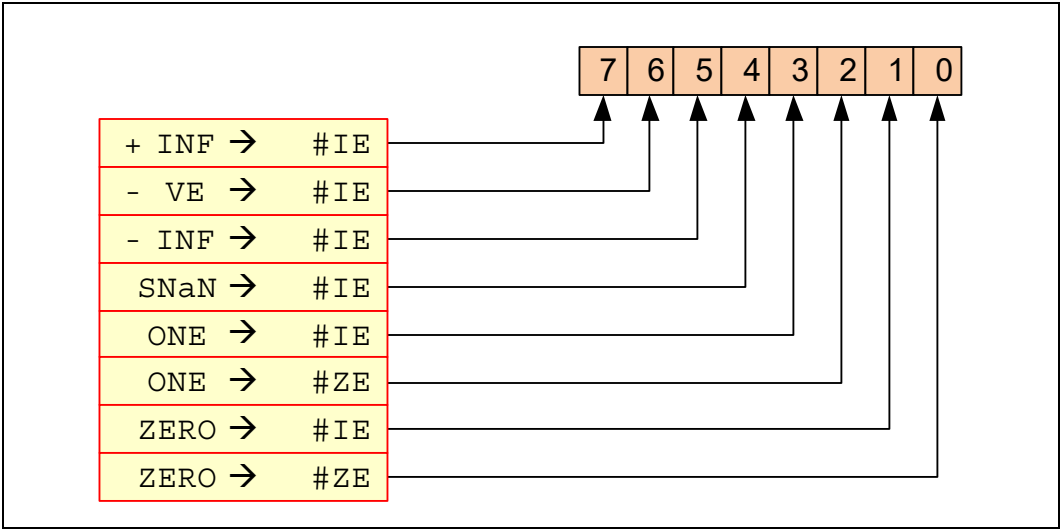


Figure 5-11. VFIXUPIMMSD Immediate Control Description

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VFIXUPIMMSD __m128d __mm_fixupimm_sd( __m128d a, __m128i tbl, int imm);  
VFIXUPIMMSD __m128d __mm_mask_fixupimm_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);  
VFIXUPIMMSD __m128d __mm_maskz_fixupimm_sd( __mmask8 k, __m128d a, __m128i tbl, int imm);  
VFIXUPIMMSD __m128d __mm_fixupimm_round_sd( __m128d a, __m128i tbl, int imm, int sae);  
VFIXUPIMMSD __m128d __mm_mask_fixupimm_round_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);  
VFIXUPIMMSD __m128d __mm_maskz_fixupimm_round_sd( __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);
```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E3.

## VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 55 /r ib VFIXUPIMMSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F	Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Perform a fix-up of the low doubleword element encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , yields an incorrect result. To deal with this, `VFIXUPIMMSS` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If `MXCSR.DAZ` is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

`MXCSR.DAZ` is used and refer to `zmm2` only (i.e. `zmm1` is not considered as zero in case `MXCSR.DAZ` is set).

`MXCSR` mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, `MXCSR.IE` or `MXCSR.ZE` might be updated.

### Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```

```

FIXUPIMM_SP (dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j = 7;
    }          ; end source special CASE(tsrc...)

; The required response from src3 table is extracted
token_response[3:0] = tbl3[3+4*j:4*j];

CASE(token_response[3:0]) {
    0000: dest[31:0] ← dest[31:0];          ; preserve content of DEST
    0001: dest[31:0] ← tsrc[31:0];          ; pass through src1 normal input value, denormal as zero
    0010: dest[31:0] ← QNaN(tsrc[31:0]);
    0011: dest[31:0] ← QNaN_Indefinite;
    0100: dest[31:0] ← -INF;
    0101: dest[31:0] ← +INF;
    0110: dest[31:0] ← tsrc.sign? -INF : +INF;
    0111: dest[31:0] ← -0;
    1000: dest[31:0] ← +0;
    1001: dest[31:0] ← -1;
    1010: dest[31:0] ← +1;
    1011: dest[31:0] ← ½;
    1100: dest[31:0] ← 90.0;
    1101: dest[31:0] ← PI/2;
    1110: dest[31:0] ← MAX_FLOAT;
    1111: dest[31:0] ← -MAX_FLOAT;
}          ; end of token_response CASE

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
    ; end fault reporting
return dest[31:0];
}          ; end of FIXUPIMM_SP()

```

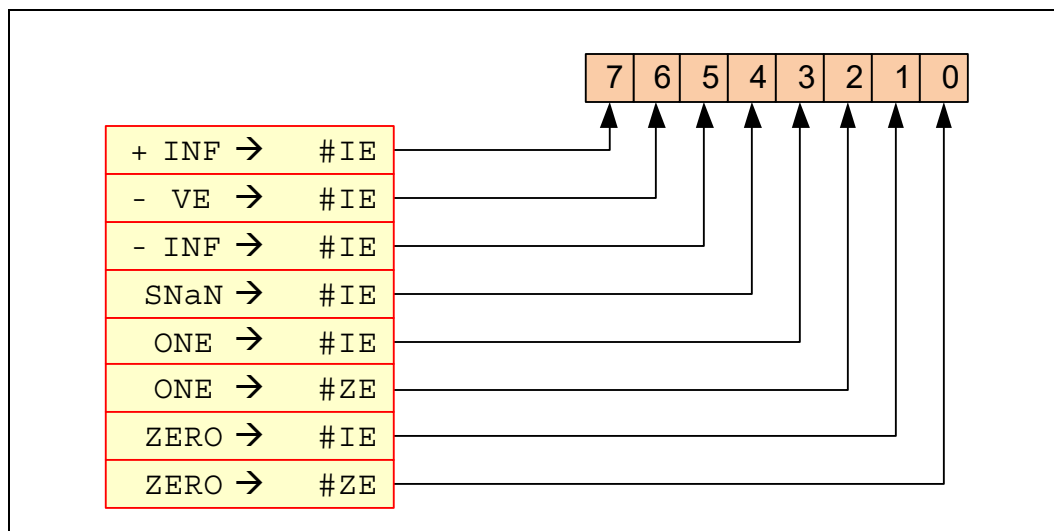
**VFIXUPIMMSS (EVEX encoded version)**

```

IF k1[0] OR *no writemask*
  THEN DEST[31:0] ← FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
ELSE
  IF *merging-masking*                ; merging-masking
    THEN *DEST[31:0] remains unchanged*
  ELSE DEST[31:0] ← 0                ; zeroing-masking
FI
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

Immediate Control Description:



**Figure 5-12. VFIXUPIMMSS Immediate Control Description**

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFIXUPIMMSS __m128 __mm_fixupimm_ss( __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_ss( __m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_ss( __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_fixupimm_round_ss( __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_round_ss( __m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_round_ss( __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E3.

## VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W1 98 /r VFMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 A8 /r VFMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1.



## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMADD132PD:** Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMADD213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMADD231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) is a ZMM register and encoded in `reg_field`. The second source operand is a ZMM register and encoded in `EVEX.vvvv`. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask `k1`.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

**Operation**

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMAADD132PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFMAADD213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFMAADD231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] + SRC2[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(DEST[i+63:i]\*SRC3[63:0] + SRC2[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(DEST[i+63:i]\*SRC3[i+63:i] + SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] + DEST[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*SRC3[63:0] + DEST[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*SRC3[i+63:i] + DEST[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD __m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_fmadd_pd(__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d _mm256_fmadd_pd(__m256d a, __m256d b, __m256d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 98 /r VFMADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 98 /r VFMADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W0 B8 /r VFMADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 98 /r VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 98 /r VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 B8 /r VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 98 /r VFMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 A8 /r VFMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 B8 /r VFMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMADD132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) is a ZMM register and encoded in `reg_field`. The second source operand is a ZMM register and encoded in `EVEX.vvvv`. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask `k1`.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.



## Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

### VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ← 4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

### VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ← 4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

### VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ← 4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

**VFMAADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

RoundFPControl(DEST[i+31:i]\*SRC3[i+31:i] + SRC2[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(DEST[i+31:i]\*SRC3[31:0] + SRC2[i+31:i])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(DEST[i+31:i]\*SRC3[i+31:i] + SRC2[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*SRC3[i+31:i] + DEST[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[31:0] + DEST[i+31:i])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[i+31:i] + DEST[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDxxxPS __m512 __mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 __mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 __mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 __mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 __mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 __mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-add computation on the low double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

**VFMADD132SD:** Multiplies the low double-precision floating-point value from the first source operand to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point values in the second source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VFMADD213SD:** Multiplies the low double-precision floating-point value from the second source operand to the low double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VFMADD231SD:** Multiplies the low double-precision floating-point value from the second source to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

**EVEX encoded version:** The low quadword element of the destination is updated according to the writemask.

**Operation**

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(DEST[63:0]\*SRC3[63:0] + SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(SRC2[63:0]\*DEST[63:0] + SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFMAADD231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                               ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

```

**VFMAADD132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← MAXVL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAXVL-1:128] ← 0

```

**VFMAADD213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAXVL-1:128] ← 0

```

**VFMAADD231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMAADDxxxSD __m128d __mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMAADDxxxSD __m128d __mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMAADDxxxSD __m128d __mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMAADDxxxSD __m128d __mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMAADDxxxSD __m128d __mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMAADDxxxSD __m128d __mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMAADDxxxSD __m128d __mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMAADDxxxSD __m128d __mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.



## VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-add computation on single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

**VFMADD132SS:** Multiplies the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

**VFMADD213SS:** Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

**VFMADD231SS:** Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE                                ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

#### VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE                                ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

**VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDxxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 _mm_fmadd_ss(__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.DDS.512.66.0F38.W1 A6 /r VFMADDSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 B6 /r VFMADDSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 96 /r VFMADDSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFMADDSUB132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg\_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm\_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFMADDSUB132PD *DEST, SRC2, SRC3*

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
DEST[MAXVL-1:128] ← 0
```

ELSEIF (VEX.256)

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192])
```

FI

#### VFMADDSUB213PD *DEST, SRC2, SRC3*

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
DEST[MAXVL-1:128] ← 0
```

ELSEIF (VEX.256)

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192])
```

FI

#### VFMADDSUB231PD *DEST, SRC2, SRC3*

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
DEST[MAXVL-1:128] ← 0
```

ELSEIF (VEX.256)

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192])
```

FI

**VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] - SRC2[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] + SRC2[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
            ELSE
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
          FI;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
  DEST[MAXVL-1:VL] ← 0

```



**VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
            ELSE
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
            ELSE
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
          FI;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
  DEST[MAXVL-1:VL] ← 0

```

**VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] - DEST[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] + DEST[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
            ELSE
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
            ELSE
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
          FI;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
  DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDSUBxxxPD __m512d __mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_mask3_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDSUBxxxPD __m256d __mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_mask_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m256d __mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add/subtract elements in zmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 A6 /r VFMADDSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 B6 /r VFMADDSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 96 /r VFMADDSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFMADDSUB132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADDSUB132PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM - 1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFMADDSUB213PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM - 1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```



**VFMADDSUB231PS DEST, SRC2, SRC3**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM - 1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
            ELSE
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                DEST[i+31:i] ←
                  RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
              ELSE
                DEST[i+31:i] ←
                  RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
              FI;
            FI
          ELSE
            IF *merging-masking* ; merging-masking
              THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
              DEST[i+31:i] ← 0
            FI
          FI;
        ENDFOR
      DEST[MAXVL-1:VL] ← 0

```

**VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI;

```

        FI
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFMAADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
    FOR j ← 0 TO KL-1
        i ← j * 32
        IF k1[j] OR *no writemask*
            THEN
                IF j *is even*
                    THEN DEST[i+31:i] ←
                        RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                    ELSE DEST[i+31:i] ←
                        RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                FI
            ELSE
                IF *merging-masking*                ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
                FI
            FI;
        ENDFOR
    DEST[MAXVL-1:VL] ← 0

```

**VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
            ELSE
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[j+31:i] - DEST[i+31:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                DEST[i+31:i] ←
                  RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
              ELSE
                DEST[i+31:i] ←
                  RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[j+31:i] + DEST[i+31:i])
              FI;
            FI
          ELSE
            IF *merging-masking* ; merging-masking
              THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
              DEST[i+31:i] ← 0
            FI
          FI;
        ENDFOR
      DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMADDSUBxxxPS __m512 __mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512 __mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxxPS __m256 __mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128 __mm_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __mm_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __mm_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m128 __mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m256 __mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.512.66.0F38.W1 97 /r VFMSUBADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 A7 /r VFMSUBADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 B7 /r VFMSUBADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**VFMSUBADD132PD:** Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD213PD:** Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD231PD:** Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.



VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### **VFMSUBADD132PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
DEST[MAXVL-1:128] ← 0
```

ELSEIF (VEX.256)

```
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
DEST[191:128] ← RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
DEST[255:192] ← RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192])
```

FI

#### **VFMSUBADD213PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
DEST[MAXVL-1:128] ← 0
```

ELSEIF (VEX.256)

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192])
```

FI

#### **VFMSUBADD231PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
DEST[MAXVL-1:128] ← 0
```

ELSEIF (VEX.256)

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
DEST[191:128] ← RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
DEST[255:192] ← RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192])
```

FI

**VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] + SRC2[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] - SRC2[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[j+63:i] + SRC2[i+63:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                DEST[i+63:i] ←
                  RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
              ELSE
                DEST[i+63:i] ←
                  RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[j+63:i] - SRC2[i+63:i])
              FI;
            FI
          ELSE
            IF *merging-masking* ; merging-masking
              THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
              DEST[i+63:i] ← 0
            FI
          FI;
        ENDFOR
      DEST[MAXVL-1:VL] ← 0

```

**VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
            ELSE
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                DEST[i+63:i] ←
                  RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
              ELSE
                DEST[i+63:i] ←
                  RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
              FI;
            FI
          ELSE
            IF *merging-masking* ; merging-masking
              THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
              DEST[i+63:i] ← 0
            FI
          FI;
        ENDFOR
      DEST[MAXVL-1:VL] ← 0

```

**VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] + DEST[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] - DEST[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
            ELSE
              DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[j+63:i] + DEST[i+63:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                DEST[i+63:i] ←
                  RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])

                ELSE
                  DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[j+63:i] - DEST[i+63:i])
                  FI;
            FI
          ELSE
            IF *merging-masking* ; merging-masking
              THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
              DEST[i+63:i] ← 0
            FI
          FI;
        ENDFOR
      DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBADDxxxPD __m512d __mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBADDxxxPD __m256d __mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm_mask_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m256d __mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.



## VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 97 /r VFMSUBADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 A7 /r VFMSUBADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 B7 /r VFMSUBADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFMSUBADD132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

## Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

### VFMSUBADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

### VFMSUBADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

### VFMSUBADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] ← RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

**VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(DEST[i+31:i]\*SRC3[i+31:i] + SRC2[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(DEST[i+31:i]\*SRC3[i+31:i] - SRC2[i+31:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
            ELSE
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
            ELSE
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
          FI;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
  DEST[MAXVL-1:VL] ← 0

```

**VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[j+31:i] ←
                RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] + SRC3[31:0])
            ELSE
              DEST[j+31:i] ←
                RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] + SRC3[j+31:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[j+31:i] ←
                RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] - SRC3[j+31:i])
            ELSE
              DEST[j+31:i] ←
                RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] - SRC3[31:0])
          FI;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[j+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[j+31:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[MAXVL-1:VL] ← 0

```

**VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*SRC3[i+31:i] + DEST[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*SRC3[i+31:i] - DEST[i+31:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0



**VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
            ELSE
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
            ELSE
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
          FI;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
  DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBADDxxxPS __m512 __mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512 __mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __mm512_mask3_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBADDxxxPS __m256 __mm256_mask_fmsubadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 __mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 __mm256_mask3_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBADDxxxPS __m128 __mm_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 __mm_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 __mm_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBADDxxxPS __m128 __mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 __mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9A /r VFMSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 AA /r VFMSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 BA /r VFMSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 9A /r VFMSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 AA /r VFMSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 BA /r VFMSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 9A /r VFMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1.
EVEX.NDS.128.66.0F38.W1 AA /r VFMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.NDS.128.66.0F38.W1 BA /r VFMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract xmm1 and put result in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 9A /r VFMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 AA /r VFMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 BA /r VFMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract ymm1 and put result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 9A /r VFMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract zmm2 and put result in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 AA /r VFMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 BA /r VFMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMSUB132PD:** Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

### VFMSUB132PD DEST, SRC2, SRC3 (VEX encoded versions)

```
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

### VFMSUB213PD DEST, SRC2, SRC3 (VEX encoded versions)

```
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

### VFMSUB231PD DEST, SRC2, SRC3 (VEX encoded versions)

```
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

**VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] - SRC2[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(DEST[i+63:i]\*SRC3[63:0] - SRC2[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(DEST[i+63:i]\*SRC3[i+63:i] - SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[63:0])

+31:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] - DEST[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*SRC3[63:0] - DEST[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(SRC2[i+63:i]\*SRC3[i+63:i] - DEST[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxPD __m512d __mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d __mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d __mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d __mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d __mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d __mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d __mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d __mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d __mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m256d __mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/E n	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W0 BA /r VFMSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 BA /r VFMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 9A /r VFMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 AA /r VFMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 BA /r VFMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMSUB132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

**Operation**

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

RoundFPControl(DEST[i+31:i]\*SRC3[i+31:i] - SRC2[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(DEST[i+31:i]\*SRC3[31:0] - SRC2[i+31:i])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(DEST[i+31:i]\*SRC3[i+31:i] - SRC2[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[31:0])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[i+31:i] - DEST[i+31:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[31:0] - DEST[i+31:i])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[i+31:i] - DEST[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxPS __m512 __mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBxxxPS __m256 __mm256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m256 __mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.



## VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

**VFMSUB132SD:** Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMSUB213SD:** Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMSUB231SD:** Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(DEST[63:0]\*SRC3[63:0] - SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

#### VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(SRC2[63:0]\*DEST[63:0] - SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                                ; zeroing-masking
            THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

```

**VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

```

**VFMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

```

**VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxSD __m128d __mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d __mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d __mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d __mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d __mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

**VFMSUB132SS:** Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFMSUB213SS:** Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFMSUB231SS:** Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE                                ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

#### VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE                                ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

**VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                               ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 _mm_fmsub_ss(__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W1 9C /r VFNMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 AC /r VFNMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 BC /r VFNMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

VFMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, "\*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).



**VFNMADD132PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFNMADD213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFNMADD231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(-(DEST[i+63:i]\*SRC3[i+63:i]) + SRC2[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(DEST[i+63:i]\*SRC3[63:0]) + SRC2[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(DEST[i+63:i]\*SRC3[i+63:i]) + SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[63:0])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ←

RoundFPControl(-(SRC2[i+63:i]\*SRC3[i+63:i]) + DEST[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*SRC3[63:0]) + DEST[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*SRC3[i+63:i]) + DEST[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

```

VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 BC /r VFNMADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 BC /r VFNMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 9C /r VFNMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 AC /r VFNMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 BC /r VFNMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFMADD132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFNMADD213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM  $\leftarrow$  2
ELSEIF (VEX.256)
    MAXNUM  $\leftarrow$  4
FI

For i = 0 to MAXNUM-1 {
    n  $\leftarrow$  32*i;
    DEST[n+31:n]  $\leftarrow$  RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128]  $\leftarrow$  0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256]  $\leftarrow$  0
FI

```

**VFNMADD231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM  $\leftarrow$  2
ELSEIF (VEX.256)
    MAXNUM  $\leftarrow$  4
FI

For i = 0 to MAXNUM-1 {
    n  $\leftarrow$  32*i;
    DEST[n+31:n]  $\leftarrow$  RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128]  $\leftarrow$  0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256]  $\leftarrow$  0
FI

```

**VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j  $\leftarrow$  0 TO KL-1
    i  $\leftarrow$  j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i]  $\leftarrow$ 
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i]  $\leftarrow$  0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL]  $\leftarrow$  0

```



**VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) + SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDIF
  DEST[MAXVL-1:VL] ← 0

```

**VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
  FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ←
        RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDIF
  DEST[MAXVL-1:VL] ← 0

```

**VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])

          ELSE
            DEST[i+31:i] ←
              RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
            FI;
        ELSE
          IF *merging-masking*           ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
          ELSE                             ; zeroing-masking
            DEST[i+31:i] ← 0
          FI
        FI;
      ENDIF
    ENDIF
  DEST[MAXVL-1:VL] ← 0

```

**VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDIF
  DEST[MAXVL-1:VL] ← 0

```

**VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
        FI;
      ELSE
        IF *merging-masking*                ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                                  ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF;
  DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFNMADDxxxPS __m512 __mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 __mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 __mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512 __mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 __mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512 __mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 __mm512_maskz_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 __mm512_mask3_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMADDxxxPS __m256 __mm256_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256 __mm256_maskz_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 __mm256_mask3_fnmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMADDxxxPS __m128 __mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxPS __m128 __mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 __mm_mask3_fnmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxPS __m128 __mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m256 __mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 9D /r VFNMADD132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 AD /r VFNMADD213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 BD /r VFNMADD231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 9D /r VFNMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 AD /r VFNMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 BD /r VFNMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and add to xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**VFNMADD132SD:** Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMADD213SD:** Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMADD231SD:** Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(-(DEST[63:0]\*SRC3[63:0]) + SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

#### VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(-(SRC2[63:0]\*DEST[63:0]) + SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(-(SRC2[63:0]\*SRC3[63:0]) + DEST[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMADD132SD DEST, SRC2, SRC3 (VEX encoded version)**

DEST[63:0]  $\leftarrow$  RoundFPControl\_MXCSR(- (DEST[63:0]\*SRC3[63:0]) + SRC2[63:0])

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMADD213SD DEST, SRC2, SRC3 (VEX encoded version)**

DEST[63:0]  $\leftarrow$  RoundFPControl\_MXCSR(- (SRC2[63:0]\*DEST[63:0]) + SRC3[63:0])

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMADD231SD DEST, SRC2, SRC3 (VEX encoded version)**

DEST[63:0]  $\leftarrow$  RoundFPControl\_MXCSR(- (SRC2[63:0]\*SRC3[63:0]) + DEST[63:0])

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFNMADDxxxSD \_\_m128d \_mm\_fnmadd\_round\_sd(\_\_m128d a, \_\_m128d b, \_\_m128d c, int r);

VFNMADDxxxSD \_\_m128d \_mm\_mask\_fnmadd\_sd(\_\_m128d a, \_\_mmask8 k, \_\_m128d b, \_\_m128d c);

VFNMADDxxxSD \_\_m128d \_mm\_maskz\_fnmadd\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b, \_\_m128d c);

VFNMADDxxxSD \_\_m128d \_mm\_mask3\_fnmadd\_sd(\_\_m128d a, \_\_m128d b, \_\_m128d c, \_\_mmask8 k);

VFNMADDxxxSD \_\_m128d \_mm\_mask\_fnmadd\_round\_sd(\_\_m128d a, \_\_mmask8 k, \_\_m128d b, \_\_m128d c, int r);

VFNMADDxxxSD \_\_m128d \_mm\_maskz\_fnmadd\_round\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b, \_\_m128d c, int r);

VFNMADDxxxSD \_\_m128d \_mm\_mask3\_fnmadd\_round\_sd(\_\_m128d a, \_\_m128d b, \_\_m128d c, \_\_mmask8 k, int r);

VFNMADDxxxSD \_\_m128d \_mm\_fnmadd\_sd (\_\_m128d a, \_\_m128d b, \_\_m128d c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 9D /r VFMADD132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 AD /r VFMADD213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 BD /r VFMADD231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 9D /r VFMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 AD /r VFMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 BD /r VFMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg\_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm\_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

#### VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0]  $\leftarrow$  RoundFPControl(-(DEST[31:0]\*SRC3[31:0]) + SRC2[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0]  $\leftarrow$  0

FI;

FI;

DEST[127:32]  $\leftarrow$  DEST[127:32]

DEST[MAXVL-1:128]  $\leftarrow$  0

#### VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0]  $\leftarrow$  RoundFPControl(-(SRC2[31:0]\*DEST[31:0]) + SRC3[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0]  $\leftarrow$  0

FI;

FI;

DEST[127:32]  $\leftarrow$  DEST[127:32]

DEST[MAXVL-1:128]  $\leftarrow$  0



**VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFNMADD132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFNMADD213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFNMADD231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFNMADDxxxSS __m128 __mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 __mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxSS __m128 __mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxSS __m128 __mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxSS __m128 __mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 __mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 __mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxSS __m128 __mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m64bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W1 9E /r VFNMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 AE /r VFNMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m64bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 BE /r VFNMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm1 and put result in zmm1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFNMSUB132PD:** Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFNMSUB213PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFNMSUB231PD:** Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM  $\leftarrow$  2
ELSEIF (VEX.256)
    MAXNUM  $\leftarrow$  4
FI
For i = 0 to MAXNUM-1 {
    n  $\leftarrow$  64*i;
    DEST[n+63:n]  $\leftarrow$  RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128]  $\leftarrow$  0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256]  $\leftarrow$  0
FI

```

**VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM  $\leftarrow$  2
ELSEIF (VEX.256)
    MAXNUM  $\leftarrow$  4
FI
For i = 0 to MAXNUM-1 {
    n  $\leftarrow$  64*i;
    DEST[n+63:n]  $\leftarrow$  RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128]  $\leftarrow$  0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256]  $\leftarrow$  0
FI

```

**VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j  $\leftarrow$  0 TO KL-1
    i  $\leftarrow$  j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i]  $\leftarrow$ 
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i]  $\leftarrow$  0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL]  $\leftarrow$  0

```

**VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) - SRC2[i+63:i])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
      RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[63:0])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
        FI;
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF
  ENDIF
  DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
  FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] ←
        RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDIF
  ENDIF
  DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) - DEST[i+63:i])
        ELSE
          DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
        FI;
      ELSE
        IF *merging-masking*                ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                                ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    FI;
  ENDFOR
  DEST[MAXVL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VFNMSUBxxxPD __m512d __mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d __mm512_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __mm512_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMSUBxxxPD __m256d __mm256_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d __mm256_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d __mm256_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m128d __mm_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d __mm_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d __mm_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxPD __m128d __mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m256d __mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

```

#### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

#### Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 BE /r VFNMSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result subtract add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 BE /r VFNMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result subtract add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 9E /r VFNMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 AE /r VFNMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 BE /r VFNMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result subtract add to zmm1 and put result in zmm1.



## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFNMSUB132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in `reg_field`. The second source operand is a YMM register and encoded in `VEX.vvvv`. The third source operand is a YMM register or a 256-bit memory location and encoded in `rm_field`.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in `reg_field`. The second source operand is a XMM register and encoded in `VEX.vvvv`. The third source operand is a XMM register or a 128-bit memory location and encoded in `rm_field`. The upper 128 bits of the YMM destination register are zeroed.

## Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI

```

**VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) - SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
        FI;
      ELSE
        IF *merging-masking*                ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                                  ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    ENDIF
  ENDIF
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
    ELSE
      IF *merging-masking*                ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                                  ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDIF
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
        ELSE
          DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
        FI;
      ELSE
        IF *merging-masking*                ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFNMSUBxxxPS __m512 __mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 __mm512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __mm512_mask3_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMSUBxxxPS __m256 __mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 __mm_mask_fnmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __mm_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __mm_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m128 __mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m256 __mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**VFNMSUB132SD:** Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMSUB213SD:** Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMSUB231SD:** Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

#### VFNMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(-(DEST[63:0]\*SRC3[63:0]) - SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

#### VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(-(SRC2[63:0]\*DEST[63:0]) - SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  RoundFPControl(-(SRC2[63:0]\*SRC3[63:0]) - DEST[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)**

DEST[63:0]  $\leftarrow$  RoundFPControl\_MXCSR(- (DEST[63:0]\*SRC3[63:0]) - SRC2[63:0])

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)**

DEST[63:0]  $\leftarrow$  RoundFPControl\_MXCSR(- (SRC2[63:0]\*DEST[63:0]) - SRC3[63:0])

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)**

DEST[63:0]  $\leftarrow$  RoundFPControl\_MXCSR(- (SRC2[63:0]\*SRC3[63:0]) - DEST[63:0])

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFNMSUBxxxSD \_\_m128d \_mm\_fnmsub\_round\_sd(\_\_m128d a, \_\_m128d b, \_\_m128d c, int r);

VFNMSUBxxxSD \_\_m128d \_mm\_mask\_fnmsub\_sd(\_\_m128d a, \_\_mmask8 k, \_\_m128d b, \_\_m128d c);

VFNMSUBxxxSD \_\_m128d \_mm\_maskz\_fnmsub\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b, \_\_m128d c);

VFNMSUBxxxSD \_\_m128d \_mm\_mask3\_fnmsub\_sd(\_\_m128d a, \_\_m128d b, \_\_m128d c, \_\_mmask8 k);

VFNMSUBxxxSD \_\_m128d \_mm\_mask\_fnmsub\_round\_sd(\_\_m128d a, \_\_mmask8 k, \_\_m128d b, \_\_m128d c, int r);

VFNMSUBxxxSD \_\_m128d \_mm\_maskz\_fnmsub\_round\_sd(\_\_mmask8 k, \_\_m128d a, \_\_m128d b, \_\_m128d c, int r);

VFNMSUBxxxSD \_\_m128d \_mm\_mask3\_fnmsub\_round\_sd(\_\_m128d a, \_\_m128d b, \_\_m128d c, \_\_mmask8 k, int r);

VFNMSUBxxxSD \_\_m128d \_mm\_fnmsub\_sd (\_\_m128d a, \_\_m128d b, \_\_m128d c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.



## VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**VFNMSUB132SS:** Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFNMSUB213SS:** Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VFNMSUB231SS:** Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

**EVEX encoded version:** The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “\*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0]  $\leftarrow$  RoundFPControl(-(DEST[31:0]\*SRC3[31:0]) - SRC2[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0]  $\leftarrow$  0

FI;

FI;

DEST[127:32]  $\leftarrow$  DEST[127:32]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 \*is a register\*

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0]  $\leftarrow$  RoundFPControl(-(SRC2[31:0]\*DEST[31:0]) - SRC3[31:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0]  $\leftarrow$  0

FI;

FI;

DEST[127:32]  $\leftarrow$  DEST[127:32]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFNMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFNMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**VFNMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] ← RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFNMSUBxxxSS __m128 __fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 __mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 __mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 __mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxSS __m128 __mm_mask_fnmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 __mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 __mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxSS __m128 __fnmsub_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFPCLASSPD—Tests Types Of a Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

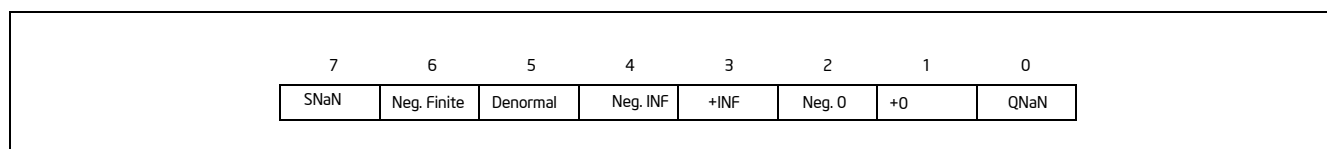
### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

The FPCLASSPD instruction checks the packed double precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX\_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-13.



**Figure 5-13. Imm8 Byte Specifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS**

**Table 5-13. Classifier Operations for VFPCLASSPD/SD/PS/SS**

Bits	Imm8[0]	Imm8[1]	Imm8[2]	Imm8[3]	Imm8[4]	Imm8[5]	Imm8[6]	Imm8[7]
Category	QNaN	PosZero	NegZero	PosINF	NegINF	Denormal	Negative	SNAN
Classifier	Checks for QNaN	Checks for +0	Checks for -0	Checks for +INF	Checks for -INF	Checks for Denormal	Checks for Negative finite	Checks for SNaN

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation**

```
CheckFPClassDP (tsrc[63:0], imm8[7:0]){
```

```
    /* Start checking the source operand for special type */
    NegNum ← tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber ← ExpAllZeros AND MantAllZeros
    SignalingBit ← tsrc[51];

    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    Nlnf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND Plnf_res ) OR
              ( imm8[4] AND Nlnf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} /* end of CheckFPClassDP() */
```

**VFPCLASSPD (EVEX Encoded versions)**

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
```

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] ← CheckFPClassDP(SRC1[63:0], imm8[7:0]);
                ELSE
                    DEST[j] ← CheckFPClassDP(SRC1[i+63:i], imm8[7:0]);
            FI;
        ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFPCLASSPD __mmask8 _mm512_fpclass_pd_mask( __m512d a, int c);
VFPCLASSPD __mmask8 _mm512_mask_fpclass_pd_mask( __mmask8 m, __m512d a, int c)
VFPCLASSPD __mmask8 _mm256_fpclass_pd_mask( __m256d a, int c)
VFPCLASSPD __mmask8 _mm256_mask_fpclass_pd_mask( __mmask8 m, __m256d a, int c)
VFPCLASSPD __mmask8 _mm_fpclass_pd_mask( __m128d a, int c)
VFPCLASSPD __mmask8 _mm_mask_fpclass_pd_mask( __mmask8 m, __m128d a, int c)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4

#UD                      If EVEX.vvvv != 1111B.

## VFPCLASSPS—Tests Types Of a Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

The FPCLASSPS instruction checks the packed single-precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX\_KL-1:16/8/4] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-13.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

CheckFPClassSP (tsrc[31:0], imm8[7:0]){

```
    /* Start checking the source operand for special type */
```

```
    NegNum ← tsrc[31];
```

```
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
```

```
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
```

```
    IF (ExpAllZeros AND MXCSR.DAZ) Then
```

```
        MantAllZeros ← 1;
```

```
    ELSIF (tsrc[22:0]=0h) Then
```

```
        MantAllZeros ← 1;
```

```
    FI;
```

```
    ZeroNumber= ExpAllZeros AND MantAllZeros
```

```
    SignalingBit= tsrc[22];
```

```
    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
```

```
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
```

```
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
```

```

Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
Nlnf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND qNaN_res ) OR ( imm8[1] AND Pzero_res ) OR
           ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND Plnf_res ) OR
           ( imm8[4] AND Nlnf_res ) OR ( imm8[5] AND Denorm_res ) OR
           ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
Return bResult;
} /* end of CheckSPClassSP() */

```

**VFPCLASSPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1) AND (SRC *is memory*)
        THEN
          DEST[j] ← CheckFPClassDP(SRC1[31:0], imm8[7:0]);
        ELSE
          DEST[j] ← CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
      FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFPCLASSPS __mmask16 _mm512_fpclass_ps_mask( __m512 a, int c);
VFPCLASSPS __mmask16 _mm512_mask_fpclass_ps_mask( __mmask16 m, __m512 a, int c)
VFPCLASSPS __mmask8 _mm256_fpclass_ps_mask( __m256 a, int c)
VFPCLASSPS __mmask8 _mm256_mask_fpclass_ps_mask( __mmask8 m, __m256 a, int c)
VFPCLASSPS __mmask8 _mm_fpclass_ps_mask( __m128 a, int c)
VFPCLASSPS __mmask8 _mm_mask_fpclass_ps_mask( __mmask8 m, __m128 a, int c)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4

#UD If EVEX.vvvv != 1111B.



## VFPClassSSD—Tests Types Of a Scalar Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W1 67 /r ib VFPClassSSD k2 {k1}, xmm2/m64, imm8	A	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

The FPCLASSSD instruction checks the low double precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX\_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-13.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

CheckFPClassDP (tsrc[63:0], imm8[7:0]){

```

NegNum ← tsrc[63];
IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
IF (ExpAllZeros AND MXCSR.DAZ) Then
    MantAllZeros ← 1;
ELSIF (tsrc[51:0]=0h) Then
    MantAllZeros ← 1;
FI;
ZeroNumber ← ExpAllZeros AND MantAllZeros
SignalingBit ← tsrc[51];

sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
Nlnf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
           ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND Plnf_res ) OR
           ( imm8[4] AND Nlnf_res ) OR ( imm8[5] AND Denorm_res ) OR
           ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
Return bResult;
} /* end of CheckFPClassDP() */
```

**VFPCCLASSSD (EVEX encoded version)**

```

IF k1[0] OR *no writemask*
    THEN DEST[0] ←
        CheckFPClassDP(SRC1[63:0], imm8[7:0])
    ELSE DEST[0] ← 0 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VFPCCLASSSD __mmask8 _mm_fpclass_sd_mask( __m128d a, int c)
VFPCCLASSSD __mmask8 _mm_mask_fpclass_sd_mask( __mmask8 m, __m128d a, int c)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E6

#UD If EVEX.vvvv != 1111B.

## VFPCCLASSSS—Tests Types Of a Scalar Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W0 67 /r VFPCCLASSSS k2 {k1}, xmm2/m32, imm8	A	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

The FPCLASSSS instruction checks the low single-precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX\_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-13.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

CheckFPClassSP (tsrc[31:0], imm8[7:0]){

```

    /* Start checking the source operand for special type */
    NegNum ← tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSIF (tsrc[22:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    Nlnf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND Plnf_res ) OR
              ( imm8[4] AND Nlnf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;

```

```
} /* end of CheckSPClassSP() */
```

**VFPCLASSSS (EVEX encoded version)**

IF k1[0] OR \*no writemask\*

THEN DEST[0] ←

CheckFPClassSP(SRC1[31:0], imm8[7:0])

ELSE DEST[0] ← 0 ; zeroing-masking only

FI;

DEST[MAX\_KL-1:1] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VFPCLASSSS \_\_mmask8 \_mm\_fpclass\_ss\_mask( \_\_m128 a, int c)

VFPCLASSSS \_\_mmask8 \_mm\_mask\_fpclass\_ss\_mask( \_\_mmask8 m, \_\_m128 a, int c)

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E6

#UD If EVEX.vvvv != 1111B.

## VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/3 2-bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 92 /r VGATHERDPD <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W1 93 /r VGATHERQPD <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W1 92 /r VGATHERDPD <i>ymm1</i> , <i>vm32x</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W1 93 /r VGATHERQPD <i>ymm1</i> , <i>vm64y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather double-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg ( <i>r</i> , <i>w</i> )	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv ( <i>r</i> , <i>w</i> )	NA

### Description

The instruction conditionally loads up to 2 or 4 double-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double-precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two double-precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four double-precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

$DEST \leftarrow SRC1$ ;  
 BASE\_ADDR: base register encoded in VSIB addressing;  
 VINDEX: the vector index register encoded by VSIB addressing;  
 SCALE: scale factor encoded by SIB:[7:6];  
 DISP: optional 1, 4 byte displacement;  
 $MASK \leftarrow SRC3$ ;

### VGATHERDPD (VEX.128 version)

```

FOR j ← 0 to 1
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  k ← j * 32;
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
  
```

### VGATHERQPD (VEX.128 version)

```

FOR j ← 0 to 1
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
  
```

**VGATHERQPD (VEX.256 version)**

```

FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

**VGATHERDPD (VEX.256 version)**

```

FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  k ← j * 32;
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```



## Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD: `__m128d _mm_i32gather_pd (double const * base, __m128i index, const int scale);`

VGATHERDPD: `__m128d _mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);`

VGATHERDPD: `__m256d _mm256_i32gather_pd (double const * base, __m128i index, const int scale);`

VGATHERDPD: `__m256d _mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int scale);`

VGATHERQPD: `__m128d _mm_i64gather_pd (double const * base, __m128i index, const int scale);`

VGATHERQPD: `__m128d _mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);`

VGATHERQPD: `__m256d _mm256_i64gather_pd (double const * base, __m256i index, const int scale);`

VGATHERQPD: `__m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);`

## SIMD Floating-Point Exceptions

None

## Other Exceptions

See Exceptions Type 12.

## VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	A	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	A	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W0 92 /r VGATHERDPS <i>ymm1</i> , <i>vm32y</i> , <i>ymm2</i>	A	V/V	AVX2	Using dword indices specified in <i>vm32y</i> , gather single-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W0 93 /r VGATHERQPS <i>xmm1</i> , <i>vm64y</i> , <i>xmm2</i>	A	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

### Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST  $\leftarrow$  SRC1;

BASE\_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK  $\leftarrow$  SRC3;

**VGATHERDPS (VEX.128 version)**

```

FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 3
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

**VGATHERQPS (VEX.128 version)**

```

FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 1
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[MAXVL-1:64] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

**VGATHERDPS (VEX.256 version)**

```

FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 7
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

**VGATHERQPS (VEX.256 version)**

```

FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VGATHERDPS: `__m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);`

VGATHERDPS: `__m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);`

VGATHERDPS: `__m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);`

VGATHERDPS: `__m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);`

VGATHERQPS: `__m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);`

VGATHERQPS: `__m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);`

VGATHERQPS: `__m128 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);`

VGATHERQPS: `__m128 _mm256_mask_i64gather_ps (__m128 src, float const * base, __m256i index, __m128 mask, const int scale);`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 12.

## VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 92 /vsib VGATHERDPS xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 92 /vsib VGATHERDPS ymm1 {k1}, vm32y	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 92 /vsib VGATHERDPS zmm1 {k1}, vm32z	A	V/V	AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 92 /vsib VGATHERDPD xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 92 /vsib VGATHERDPD ymm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 92 /vsib VGATHERDPD zmm1 {k1}, vm32y	A	V/V	AVX512F	Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

### Description

A set of single-precision/double-precision floating-point memory locations pointed by base address `BASE_ADDR` and index vector `V_INDEX` with scale `SCALE` are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} \times N$  and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VGATHERDPS (EVEX encoded version)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

    i  $\leftarrow$  j \* 32

    IF k1[j]

        THEN DEST[i+31:i]  $\leftarrow$

            MEM[BASE\_ADDR +

                SignExtend(VINDEX[i+31:i]) \* SCALE + DISP]

        k1[j]  $\leftarrow$  0

    ELSE \*DEST[i+31:i]  $\leftarrow$  remains unchanged\*

    FI;

ENDFOR

k1[MAX\_KL-1:KL]  $\leftarrow$  0

DEST[MAXVL-1:VL]  $\leftarrow$  0

#### VGATHERDPD (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j  $\leftarrow$  0 TO KL-1

    i  $\leftarrow$  j \* 64

    k  $\leftarrow$  j \* 32

    IF k1[j]

        THEN DEST[i+63:i]  $\leftarrow$  MEM[BASE\_ADDR +

            SignExtend(VINDEX[k+31:k]) \* SCALE + DISP]

        k1[j]  $\leftarrow$  0

    ELSE \*DEST[i+63:i]  $\leftarrow$  remains unchanged\*

    FI;

ENDFOR

k1[MAX\_KL-1:KL]  $\leftarrow$  0

DEST[MAXVL-1:VL]  $\leftarrow$  0



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGATHERDPD __m512d __mm512_i32gather_pd( __m256i vdx, void * base, int scale);
VGATHERDPD __m512d __mm512_mask_i32gather_pd(__m512d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d __mm256_mask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d __mm_mask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_i32gather_ps( __m512i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 __mm256_mask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128 __mm_mask_i32gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

**VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z	A	V/V	AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 93 /vsib VGATHERQPD xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z	A	V/V	AVX512F	Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

**Description**

A set of 8 single-precision/double-precision floating-point memory locations pointed by base address `BASE_ADDR` and index vector `V_INDEX` with scale `SCALE` are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} \times N$  and alignment rules.  $N$  is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector `zmm1` is the same as index vector `VINDEX`. The instruction will #UD fault if the `k0` mask register is specified.

### Operation

`BASE_ADDR` stands for the memory operand base address (a GPR); may not exist

`VINDEX` stands for the memory operand vector of indices (a ZMM register)

`SCALE` stands for the memory operand scalar (1, 2, 4 or 8)

`DISP` is the optional 1, 2 or 4 byte displacement

### VGATHERQPS (EVEX encoded version)

$(KL, VL) = (2, 64), (4, 128), (8, 256)$

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

$k \leftarrow j * 64$

    IF  $k1[j]$  OR \*no writemask\*

        THEN  $\text{DEST}[i+31:i] \leftarrow$

$\text{MEM}[\text{BASE\_ADDR} + (\text{VINDEX}[k+63:k]) * \text{SCALE} + \text{DISP}]$

$k1[j] \leftarrow 0$

        ELSE  $\text{DEST}[i+31:i] \leftarrow$  remains unchanged\*

    FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$

$\text{DEST}[\text{MAXVL}-1:VL/2] \leftarrow 0$

### VGATHERQPD (EVEX encoded version)

$(KL, VL) = (2, 128), (4, 256), (8, 512)$

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

    IF  $k1[j]$  OR \*no writemask\*

        THEN  $\text{DEST}[i+63:i] \leftarrow \text{MEM}[\text{BASE\_ADDR} + (\text{VINDEX}[i+63:i]) * \text{SCALE} + \text{DISP}]$

$k1[j] \leftarrow 0$

        ELSE  $\text{DEST}[i+63:i] \leftarrow$  remains unchanged\*

    FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$

$\text{DEST}[\text{MAXVL}-1:VL] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGATHERQPD __m512d _mm512_i64gather_pd( __m512i vdx, void * base, int scale);
VGATHERQPD __m512d _mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d _mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d _mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_i64gather_ps( __m512i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 _mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 _mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

## VGETEXPPD—Convert Exponents of Packed DP FP Values to DP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 42 /r VGETEXPPD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W1 42 /r VGETEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W1 42 /r VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Extracts the biased exponents from the normalized DP FP representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double-precision FP value and written to the corresponding qword elements of the destination operand (the first operand) as DP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-14.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for the greatest integer not exceeding real number x.

**Table 5-14. VGETEXPPD/SD Special Cases**

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	No Exceptions
$0 <  \text{src1}  < \text{INF}$	$\text{floor}(\log_2( \text{src1} ))$	
$ \text{src1}  = +\text{INF}$	+INF	
$ \text{src1}  = 0$	-INF	

**Operation**

NormalizeExpTinyDPFP(SRC[63:0])

```
{
    // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit ← 0;
    Dst.exp ← 1;
    Dst.fraction ← SRC[51:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit ← Dst.fraction[51];          // Get the fraction MSB
        Dst.fraction ← Dst.fraction << 1;    // One bit shift left
        Dst.exp--;                          // Decrement the exponent
    }
    Dst.fraction ← 0;                        // zero out fraction bits
    Dst.sign ← 1;                            // Return negative sign
    TMP[63:0] ← MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
    Return (TMP[63:0]);
}
```

ConvertExpDPFP(SRC[63:0])

```
{
    Src.sign ← 0;                          // Zero out sign bit
    Src.exp ← SRC[62:52];
    Src.fraction ← SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (SRC = +INF) Return (SRC);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
}
ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
{
    IF ((Src.exp = 0) AND (Src.fraction != 0))
    {
        TMP[63:0] ← NormalizeExpTinyDPFP(SRC[63:0]);    // Get Normalized Exponent
        Set #DE
    }
    ELSE // exponent value is correct
    {
        TMP[63:0] ← (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction);
    }
    TMP ← SAR(TMP, 52);          // Shift Arithmetic Right
    TMP ← TMP - 1023;           // Subtract Bias
    Return CvtI2D(TMP);         // Convert INT to Double-Precision FP number
}
}
```

**VGETEXPPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN

DEST[i+63:i] ←

ConvertExpDPFP(SRC[63:0])

ELSE

DEST[i+63:i] ←

ConvertExpDPFP(SRC[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETEXPPD \_\_m512d \_mm512\_getexp\_pd(\_\_m512d a);

VGETEXPPD \_\_m512d \_mm512\_mask\_getexp\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VGETEXPPD \_\_m512d \_mm512\_maskz\_getexp\_pd(\_\_mmask8 k, \_\_m512d a);

VGETEXPPD \_\_m512d \_mm512\_getexp\_round\_pd(\_\_m512d a, int sae);

VGETEXPPD \_\_m512d \_mm512\_mask\_getexp\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int sae);

VGETEXPPD \_\_m512d \_mm512\_maskz\_getexp\_round\_pd(\_\_mmask8 k, \_\_m512d a, int sae);

VGETEXPPD \_\_m256d \_mm256\_getexp\_pd(\_\_m256d a);

VGETEXPPD \_\_m256d \_mm256\_mask\_getexp\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a);

VGETEXPPD \_\_m256d \_mm256\_maskz\_getexp\_pd(\_\_mmask8 k, \_\_m256d a);

VGETEXPPD \_\_m128d \_mm\_getexp\_pd(\_\_m128d a);

VGETEXPPD \_\_m128d \_mm\_mask\_getexp\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VGETEXPPD \_\_m128d \_mm\_maskz\_getexp\_pd(\_\_mmask8 k, \_\_m128d a);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 42 /r VGETEXPPS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W0 42 /r VGETEXPPS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	A	V/V	AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Extracts the biased exponents from the normalized SP FP representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision FP value and written to the corresponding dword elements of the destination operand (the first operand) as SP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor**(x) stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

**Table 5-15. VGETEXPPS/SS Special Cases**

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	No Exceptions
$0 <  \text{src1}  < \text{INF}$	$\text{floor}(\log_2( \text{src1} ))$	
$ \text{src1}  = +\text{INF}$	+INF	
$ \text{src1}  = 0$	-INF	



Figure 5-14 illustrates the VGETEXPPS functionality on input values with normalized representation.

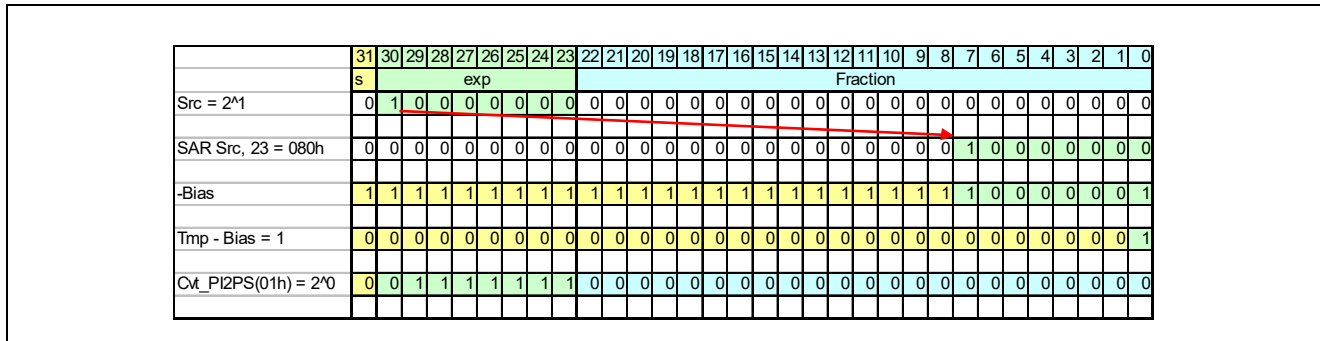


Figure 5-14. VGETEXPPS Functionality On Normal Input values

### Operation

NormalizeExpTinySPFP(SRC[31:0])

```
{
    // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit ← 0;
    Dst.exp ← 1;
    Dst.fraction ← SRC[22:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit ← Dst.fraction[22];    // Get the fraction MSB
        Dst.fraction ← Dst.fraction << 1;    // One bit shift left
        Dst.exp--;    // Decrement the exponent
    }
    Dst.fraction ← 0;    // zero out fraction bits
    Dst.sign ← 1;    // Return negative sign
    TMP[31:0] ← MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
    Return (TMP[31:0]);
}
```

ConvertExpSPFP(SRC[31:0])

```
{
    Src.sign ← 0;    // Zero out sign bit
    Src.exp ← SRC[30:23];
    Src.fraction ← SRC[22:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF (SRC = SNAN) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (SRC = +INF) Return (SRC);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
}
ELSE    // check if denormal operand (notice that MXCSR.DAZ = 0)
{
```

```

    IF ((Src.exp = 0) AND (Src.fraction != 0))
    {
        TMP[31:0] ← NormalizeExpTinySPFP(SRC[31:0]);    // Get Normalized Exponent
        Set #DE
    }
    ELSE    // exponent value is correct
    {
        TMP[31:0] ← (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction);
    }
    TMP ← SAR(TMP, 23);    // Shift Arithmetic Right
    TMP ← TMP - 127;    // Subtract Bias
    Return CvtI2D(TMP);    // Convert INT to Single-Precision FP number
}
}

```

**VGETEXPPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN

DEST[i+31:i] ←

ConvertExpSPFP(SRC[31:0])

ELSE

DEST[i+31:i] ←

ConvertExpSPFP(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGETEXPPS __m512 _mm512_getexp_ps(__m512 a);
VGETEXPPS __m512 _mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_maskz_getexp_ps(__mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_getexp_round_ps(__m512 a, int sae);
VGETEXPPS __m512 _mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae);
VGETEXPPS __m512 _mm512_maskz_getexp_round_ps(__mmask16 k, __m512 a, int sae);
VGETEXPPS __m256 _mm256_getexp_ps(__m256 a);
VGETEXPPS __m256 _mm256_mask_getexp_ps(__m256 s, __mmask8 k, __m256 a);
VGETEXPPS __m256 _mm256_maskz_getexp_ps(__mmask8 k, __m256 a);
VGETEXPPS __m128 _mm_getexp_ps(__m128 a);
VGETEXPPS __m128 _mm_mask_getexp_ps(__m128 s, __mmask8 k, __m128 a);
VGETEXPPS __m128 _mm_maskz_getexp_ps(__mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E2.

#UD                      If EVEX.vvvv != 1111B.

VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 43 /r VGETEXPSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	A	V/V	AVX512F	Convert the biased exponent (bits 62:52) of the low double-precision floating-point value in xmm3/m64 to a DP FP value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Extracts the biased exponent from the normalized DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location. The low quadword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-14.

The formula is:

GETEXP(x) = floor(log<sub>2</sub>(|x|))

Notation floor(x) stands for maximal integer not exceeding real number x.

Operation

// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

VGETEXPSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
  THEN DEST[63:0] ←
    ConvertExpDPFP(SRC2[63:0])
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[63:0] remains unchanged*
    ELSE                           ; zeroing-masking
      DEST[63:0] ← 0
FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGETEXPSD __m128d _mm_getexp_sd( __m128d a, __m128d b);
VGETEXPSD __m128d _mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_maskz_getexp_sd( __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_getexp_round_sd( __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_maskz_getexp_round_sd( __mmask8 k, __m128d a, __m128d b, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E3.

## VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	A	V/V	AVX512F	Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a SP FP value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Extracts the biased exponent from the normalized SP FP representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision FP value and written to the destination operand (the first operand) as SP FP numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location. The the low doubleword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-15.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

### Operation

// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

#### VGETEXPSS (EVEX encoded version)

```

IF k1[0] OR *no writemask*
  THEN DEST[31:0] ←
    ConvertExpDPFP(SRC2[31:0])
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[31:0] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[31:0] ← 0
  FI
FI;
ENDFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGETEXPSS __m128 _mm_getexp_ss( __m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss( __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss( __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss( __mmask8 k, __m128 a, __m128 b, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E3.

## VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 26 /r ib VGETMANTPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W1 26 /r ib VGETMANTPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F	Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Convert double-precision floating values in the source operand (the second operand) to DP FP values with the mantissa normalization and sign control specified by the *imm8* byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask *k1*. The normalized mantissa is specified by *interv* (*imm8*[1:0]) and the sign control (*sc*) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

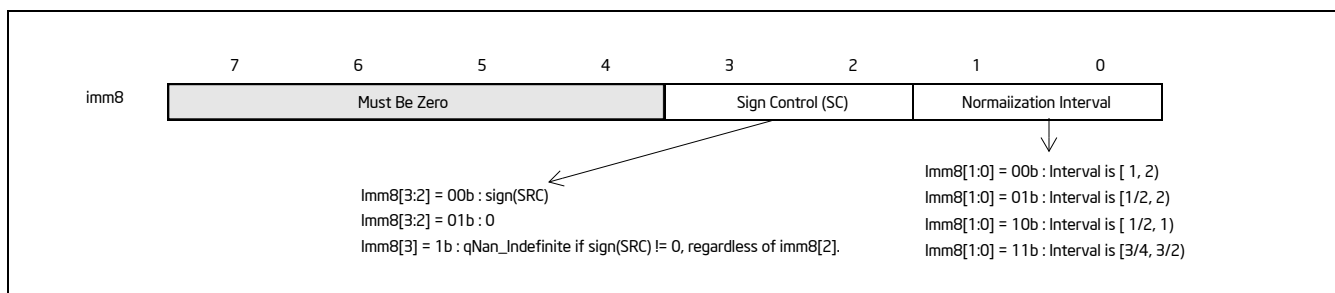


Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input DP FP value *x*, The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k / x.\text{significand}$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent *k* depends on the interval range defined by *interv* and whether the exponent of the source is even or odd. The sign of the final result is determined by *sc* and the source sign.



If  $\text{interv} \neq 0$  then  $k = -1$ , otherwise  $k = 0$ . The encoded value of  $\text{imm8}[1:0]$  and sign control are shown in Figure 5-15.

Each converted DP FP result is encoded according to the sign control, the unbiased exponent  $k$  (adding bias) and a mantissa normalized to the range specified by  $\text{interv}$ .

The `GetMant()` function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register  $k1$  are computed and stored into the destination. Elements in  $\text{zmm1}$  with the corresponding bit clear in  $k1$  retain their previous values.

Note: `EVEX.vvvv` is reserved and must be `1111b`; otherwise instructions will `#UD`.

**Table 5-16. GetMant() Special Float Values Behavior**

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore <i>interv</i> If (SRC = SNaN) then #IE
$+\infty$	1.0	Ignore <i>interv</i>
+0	1.0	Ignore <i>interv</i>
-0	IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore <i>interv</i>
$-\infty$	IF (SC[1]) THEN {QNaN_Indefinite} ELSE { IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore <i>interv</i> If (SC[1]) then #IE
negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)	If (SC[1]) then #IE

### Operation

```

GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[63];           // Get sign bit
    Dst.exp ← SRC[62:52]; // Get original exponent value
    Dst.fraction ← SRC[51:0]; // Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 07FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 07FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {
        IF (SRC = SNaN) {Set #IE;}
        Return QNaN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand))
    {
        Dst.exp ← 03FFh;           // Override exponent with BIAS
        Return ((Dst.sign << 63) | (Dst.exp << 52) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[63] = 1) AND SignCtrl[1])
    {
        Set #IE;
        Return QNaN_Indefinite;
    }
}

```

```

// Checking for denormal operands
IF (DenormOperand)
{
  IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
  ELSE
  {
    // Jbit is the hidden integral bit. Zero in case of denormal operand.
    Src.Jbit ← 0; // Zero Src Jbit
    Dst.exp ← 03FFh; // Override exponent with BIAS
    WHILE (Src.Jbit = 0) { // normalize mantissa
      Src.Jbit ← Dst.fraction[51]; // Get the fraction MSB
      Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
      Dst.exp--; // Adjust the exponent
    }
    SET #DE; // Set DE bit
  }
} // At this point, Dst.fraction is normalized.
// Checking for exponent response
Unbiased.exp ← Dst.exp - 03FFh; // subtract the bias from exponent
IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
SignalingBit ← Dst.fraction[51];
CASE (interv[1:0])
  00: Dst.exp ← 03FFh; // This is the bias
  01: Dst.exp ← (IsOddExp) ? 03FEh : 03FFh; // either bias-1, or bias
  10: Dst.exp ← 03FEh; // bias-1
  11: Dst.exp ← (SignalingBit) ? 03FEh : 03FFh; // either bias-1, or bias
ESAC
// At this point Dst.exp has the correct result. Form the final destination
DEST[63:0] ← (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
Return (DEST);
}

```

**VGETMANTPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

SignCtrl[1:0] ← IMM8[3:2];

Interv[1:0] ← IMM8[1:0];

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC \*is memory\*)

        THEN

          DEST[i+63:i] ← GetNormalizedMantissaDP(SRC[63:0], SignCtrl, Interv)

        ELSE

          DEST[i+63:i] ← GetNormalizedMantissaDP(SRC[i+63:i], SignCtrl, Interv)

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+63:i] remains unchanged\*

      ELSE ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGETMANTPD __m512d _mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d _mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);

```

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E2.

#UD                      If EVEX.vvvv != 1111B.

**VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 26 /r ib VGETMANTPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector xmm2/m128/m32bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W0 26 /r ib VGETMANTPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector ymm2/m256/m32bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W0 26 /r ib VGETMANTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F	Get normalized mantissa from float32 vector zmm2/m512/m32bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

**Description**

Convert single-precision floating values in the source operand (the second operand) to SP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input SP FP value  $x$ , The conversion operation is:

$$GetMant(x) = \pm 2^k / x.significand/$$

where:

$$1 \leq |x.significand| < 2$$

Unbiased exponent  $k$  depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then  $k = -1$ , otherwise  $K = 0$ . The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted SP FP result is encoded according to the sign control, the unbiased exponent  $k$  (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

## Operation

```

GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[31];           // Get sign bit
    Dst.exp ← SRC[30:23]; // Get original exponent value
    Dst.fraction ← SRC[22:0]; // Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 0FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 0FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {
        IF (SRC = SNaN) {Set #IE;}
        Return QNaN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand))
    {
        Dst.exp ← 07Fh; // Override exponent with BIAS
        Return ((Dst.sign << 31) | (Dst.exp << 23) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[31] = 1) AND SignCtrl[1])
    {
        Set #IE;
        Return QNaN_Indefinite;
    }
    // Checking for denormal operands
    IF (DenormOperand)
    {
        IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
        ELSE
        {
            // Jbit is the hidden integral bit. Zero in case of denormal operand.
            Src.Jbit ← 0; // Zero Src Jbit
            Dst.exp ← 07Fh; // Override exponent with BIAS
            WHILE (Src.Jbit = 0) { // normalize mantissa
                Src.Jbit ← Dst.fraction[22]; // Get the fraction MSB
                Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
                Dst.exp--; // Adjust the exponent
            }
            SET #DE; // Set DE bit
        }
    }
    // At this point, Dst.fraction is normalized.
    // Checking for exponent response
    Unbiased.exp ← Dst.exp - 07Fh; // subtract the bias from exponent
    IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
    SignalingBit ← Dst.fraction[22];
    CASE (Interv[1:0])
        00: Dst.exp ← 07Fh; // This is the bias
        01: Dst.exp ← (IsOddExp) ? 07Eh : 07Fh; // either bias-1, or bias
        10: Dst.exp ← 07Eh; // bias-1
        11: Dst.exp ← (SignalingBit) ? 07Eh : 07Fh; // either bias-1, or bias
    ESAC

    // Form the final destination
    DEST[31:0] ← (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
}

```

```

    Return (DEST);
}

```

**VGETMANTPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SignCtrl[1:0] ← IMM8[3:2];

Interv[1:0] ← IMM8[1:0];

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC \*is memory\*)

        THEN

          DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[31:0], SignCtrl, Interv)

        ELSE

          DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[j+31:i], SignCtrl, Interv)

      FI;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[j+31:i] remains unchanged\*

      ELSE ; zeroing-masking

        DEST[j+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VGETMANTPS \_\_m512 \_\_mm512\_getmant\_ps( \_\_m512 a, enum intv, enum sgn);

VGETMANTPS \_\_m512 \_\_mm512\_mask\_getmant\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, enum intv, enum sgn);

VGETMANTPS \_\_m512 \_\_mm512\_maskz\_getmant\_ps(\_\_mmask16 k, \_\_m512 a, enum intv, enum sgn);

VGETMANTPS \_\_m512 \_\_mm512\_getmant\_round\_ps( \_\_m512 a, enum intv, enum sgn, int r);

VGETMANTPS \_\_m512 \_\_mm512\_mask\_getmant\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, enum intv, enum sgn, int r);

VGETMANTPS \_\_m512 \_\_mm512\_maskz\_getmant\_round\_ps(\_\_mmask16 k, \_\_m512 a, enum intv, enum sgn, int r);

VGETMANTPS \_\_m256 \_\_mm256\_getmant\_ps( \_\_m256 a, enum intv, enum sgn);

VGETMANTPS \_\_m256 \_\_mm256\_mask\_getmant\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, enum intv, enum sgn);

VGETMANTPS \_\_m256 \_\_mm256\_maskz\_getmant\_ps( \_\_mmask8 k, \_\_m256 a, enum intv, enum sgn);

VGETMANTPS \_\_m128 \_\_mm\_getmant\_ps( \_\_m128 a, enum intv, enum sgn);

VGETMANTPS \_\_m128 \_\_mm\_mask\_getmant\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, enum intv, enum sgn);

VGETMANTPS \_\_m128 \_\_mm\_maskz\_getmant\_ps( \_\_mmask8 k, \_\_m128 a, enum intv, enum sgn);

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEEX.NDS.LIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F	Extract the normalized mantissa of the low float64 element in xmm3/m64 using <i>imm8</i> for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Convert the double-precision floating values in the low quadword element of the second source operand (the third operand) to DP FP value with the mantissa normalization and sign control specified by the *imm8* byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by *interv* (*imm8*[1:0]) and the sign control (*sc*) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$\text{GetMant}(x) = \pm 2^k / x.\text{significand}$$

where:

$$1 \leq |x.\text{significand}| < 2$$

Unbiased exponent *k* depends on the interval range defined by *interv* and whether the exponent of the source is even or odd. The sign of the final result is determined by *sc* and the source sign.

If *interv* != 0 then *k* = -1, otherwise *K* = 0. The encoded value of *imm8*[1:0] and sign control are shown in Figure 5-15.

The converted DP FP result is encoded according to the sign control, the unbiased exponent *k* (adding bias) and a mantissa normalized to the range specified by *interv*.

The *GetMant()* function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

**Operation**

// GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

**VGETMANTSD (EVEX encoded version)**

```

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ←
        GetNormalizedMantissaDP(SRC2[63:0], SignCtrl, Interv)
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE                                  ; zeroing-masking
            DEST[63:0] ← 0
        FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGETMANTSD __m128d _mm_getmant_sd( __m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd( __mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd( __m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);

```

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E3.



## VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEEX.NDS.LIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F	Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to SP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$GetMant(x) = \pm 2^k / x.significand/$$

where:

$$1 \leq |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-16 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

**Operation**

// GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

**VGETMANTSS (EVEX encoded version)**

```

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ←
        GetNormalizedMantissaSP(SRC2[31:0], SignCtrl, Interv)
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[31:0] ← 0
        FI
FI;
DEST[127:32] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGETMANTSS __m128 __mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 __mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 __mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 __mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 __mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 __mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);

```

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E3.

## VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX	Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.
EVEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	AVX512VL AVX512F	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 18 /r ib VINSERTF32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	C	V/V	AVX512F	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.256.66.0F3A.W1 18 /r ib VINSERTF64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	AVX512VL AVX512DQ	Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 18 /r ib VINSERTF64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	B	V/V	AVX512DQ	Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 1A /r ib VINSERTF32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	D	V/V	AVX512DQ	Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 1A /r ib VINSERTF64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	C	V/V	AVX512F	Insert 256 bits of packed double-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

VINSERTF128/VINSERTF32x4 and VINSERTF64x2 insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF64x2: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF32x8 and VINSERTF64x4 inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.

**Operation****VINSERTF32x4 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VINSERTF64x2 (EVEX encoded versions)**

(KL, VL) = (4, 256), (8, 512)

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

```

        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

#### VINSERTF32x8 (EVEX.U1.512 encoded version)

```

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

```

```

FOR j ← 0 TO 15
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

#### VINSERTF64x4 (EVEX.512 encoded version)

```

VL = 512
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

```

```

FOR j ← 0 TO 7
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VINSERTF128 (VEX encoded version)**

TEMP[255:0] ← SRC1[255:0]

CASE (imm8[0]) OF

0: TEMP[127:0] ← SRC2[127:0]

1: TEMP[255:128] ← SRC2[127:0]

ESAC

DEST ← TEMP

**Intel C/C++ Compiler Intrinsic Equivalent**

VINSERTF32x4 \_\_m512 \_\_mm512\_insertf32x4(\_\_m512 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m512 \_\_mm512\_mask\_insertf32x4(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m512 \_\_mm512\_maskz\_insertf32x4(\_\_mmask16 k, \_\_m512 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m256 \_\_mm256\_insertf32x4(\_\_m256 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m256 \_\_mm256\_mask\_insertf32x4(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, \_\_m128 b, int imm);

VINSERTF32x4 \_\_m256 \_\_mm256\_maskz\_insertf32x4(\_\_mmask8 k, \_\_m256 a, \_\_m128 b, int imm);

VINSERTF32x8 \_\_m512 \_\_mm512\_insertf32x8(\_\_m512 a, \_\_m256 b, int imm);

VINSERTF32x8 \_\_m512 \_\_mm512\_mask\_insertf32x8(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, \_\_m256 b, int imm);

VINSERTF32x8 \_\_m512 \_\_mm512\_maskz\_insertf32x8(\_\_mmask16 k, \_\_m512 a, \_\_m256 b, int imm);

VINSERTF64x2 \_\_m512d \_\_mm512\_insertf64x2(\_\_m512d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m512d \_\_mm512\_mask\_insertf64x2(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m512d \_\_mm512\_maskz\_insertf64x2(\_\_mmask8 k, \_\_m512d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m256d \_\_mm256\_insertf64x2(\_\_m256d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m256d \_\_mm256\_mask\_insertf64x2(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, \_\_m128d b, int imm);

VINSERTF64x2 \_\_m256d \_\_mm256\_maskz\_insertf64x2(\_\_mmask8 k, \_\_m256d a, \_\_m128d b, int imm);

VINSERTF64x4 \_\_m512d \_\_mm512\_insertf64x4(\_\_m512d a, \_\_m256d b, int imm);

VINSERTF64x4 \_\_m512d \_\_mm512\_mask\_insertf64x4(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m256d b, int imm);

VINSERTF64x4 \_\_m512d \_\_mm512\_maskz\_insertf64x4(\_\_mmask8 k, \_\_m512d a, \_\_m256d b, int imm);

VINSERTF128 \_\_m256 \_\_mm256\_insertf128\_ps(\_\_m256 a, \_\_m128 b, int offset);

VINSERTF128 \_\_m256d \_\_mm256\_insertf128\_pd(\_\_m256d a, \_\_m128d b, int offset);

VINSERTF128 \_\_m256i \_\_mm256\_insertf128\_si256(\_\_m256i a, \_\_m128i b, int offset);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

## VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX2	Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.
EVEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	AVX512VL AVX512F	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 38 /r ib VINSERTI32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	C	V/V	AVX512F	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.256.66.0F3A.W1 38 /r ib VINSERTI64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	AVX512VL AVX512DQ	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 38 /r ib VINSERTI64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	B	V/V	AVX512DQ	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 3A /r ib VINSERTI32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	D	V/V	AVX512DQ	Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 3A /r ib VINSERTI64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	C	V/V	AVX512F	Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

### Description

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.

**Operation****VINSERTI32x4 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VINSERTI64x2 (EVEX encoded versions)**

(KL, VL) = (4, 256), (8, 512)

TEMP\_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] ← SRC2[127:0]

1: TMP\_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] ← SRC2[127:0]

01: TMP\_DEST[255:128] ← SRC2[127:0]

10: TMP\_DEST[383:256] ← SRC2[127:0]

11: TMP\_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE



```

        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

#### VINSERTI32x8 (EVEX.U1.512 encoded version)

```

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

FOR j ← 0 TO 15
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

#### VINSERTI64x4 (EVEX.512 encoded version)

```

VL = 512
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

FOR j ← 0 TO 7
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VINSERTI128**

TEMP[255:0] ← SRC1[255:0]  
CASE (imm8[0]) OF  
  0: TEMP[127:0] ← SRC2[127:0]  
  1: TEMP[255:128] ← SRC2[127:0]  
ESAC  
DEST ← TEMP

**Intel C/C++ Compiler Intrinsic Equivalent**

VINSERTI32x4 \_\_mm512i\_inserti32x4(\_\_m512i a, \_\_m128i b, int imm);  
VINSERTI32x4 \_\_mm512i\_mask\_inserti32x4(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m128i b, int imm);  
VINSERTI32x4 \_\_mm512i\_maskz\_inserti32x4(\_\_mmask16 k, \_\_m512i a, \_\_m128i b, int imm);  
VINSERTI32x4 \_\_m256i\_mm256\_inserti32x4(\_\_m256i a, \_\_m128i b, int imm);  
VINSERTI32x4 \_\_m256i\_mm256\_mask\_inserti32x4(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i b, int imm);  
VINSERTI32x4 \_\_m256i\_mm256\_maskz\_inserti32x4(\_\_mmask8 k, \_\_m256i a, \_\_m128i b, int imm);  
VINSERTI32x8 \_\_m512i\_mm512\_inserti32x8(\_\_m512i a, \_\_m256i b, int imm);  
VINSERTI32x8 \_\_m512i\_mm512\_mask\_inserti32x8(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m256i b, int imm);  
VINSERTI32x8 \_\_m512i\_mm512\_maskz\_inserti32x8(\_\_mmask16 k, \_\_m512i a, \_\_m256i b, int imm);  
VINSERTI64x2 \_\_m512i\_mm512\_inserti64x2(\_\_m512i a, \_\_m128i b, int imm);  
VINSERTI64x2 \_\_m512i\_mm512\_mask\_inserti64x2(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m128i b, int imm);  
VINSERTI64x2 \_\_m512i\_mm512\_maskz\_inserti64x2(\_\_mmask8 k, \_\_m512i a, \_\_m128i b, int imm);  
VINSERTI64x2 \_\_m256i\_mm256\_inserti64x2(\_\_m256i a, \_\_m128i b, int imm);  
VINSERTI64x2 \_\_m256i\_mm256\_mask\_inserti64x2(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m128i b, int imm);  
VINSERTI64x2 \_\_m256i\_mm256\_maskz\_inserti64x2(\_\_mmask8 k, \_\_m256i a, \_\_m128i b, int imm);  
VINSERTI64x4 \_\_mm512\_inserti64x4(\_\_m512i a, \_\_m256i b, int imm);  
VINSERTI64x4 \_\_mm512\_mask\_inserti64x4(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m256i b, int imm);  
VINSERTI64x4 \_\_mm512\_maskz\_inserti64x4(\_\_mmask m, \_\_m512i a, \_\_m256i b, int imm);  
VINSERTI128 \_\_m256i\_mm256\_insertf128\_si256(\_\_m256i a, \_\_m128i b, int offset);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD                      If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

## VMASKMOV—Conditional SIMD Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 2C /r VMASKMOVPS <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX	Conditionally load packed single-precision values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2C /r VMASKMOVPS <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX	Conditionally load packed single-precision values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 2D /r VMASKMOVPD <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX	Conditionally load packed double-precision values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2D /r VMASKMOVPD <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX	Conditionally load packed double-precision values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 2E /r VMASKMOVPS <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX	Conditionally store packed single-precision values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2E /r VMASKMOVPS <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX	Conditionally store packed single-precision values from <i>ymm2</i> using mask in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 2F /r VMASKMOVPD <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX	Conditionally store packed double-precision values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 2F /r VMASKMOVPD <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX	Conditionally store packed double-precision values from <i>ymm2</i> using mask in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

### Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm\_field, and the destination register is encoded in reg\_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg\_field, and the destination memory location is encoded in rm\_field.

## Operation

### VMASKMOVPS - 128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[MAXVL-1:128] ← 0
```

### VMASKMOVPS - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

### VMASKMOVPD - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[MAXVL-1:128] ← 0
```

### VMASKMOVPD - 256-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

### VMASKMOVPS - 128-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
```

### VMASKMOVPS - 256-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ← SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
```

**VMASKMOVPD - 128-bit store**

IF (SRC1[63]) DEST[63:0]  $\leftarrow$  SRC2[63:0]  
 IF (SRC1[127]) DEST[127:64]  $\leftarrow$  SRC2[127:64]

**VMASKMOVPD - 256-bit store**

IF (SRC1[63]) DEST[63:0]  $\leftarrow$  SRC2[63:0]  
 IF (SRC1[127]) DEST[127:64]  $\leftarrow$  SRC2[127:64]  
 IF (SRC1[191]) DEST[191:128]  $\leftarrow$  SRC2[191:128]  
 IF (SRC1[255]) DEST[255:192]  $\leftarrow$  SRC2[255:192]

**Intel C/C++ Compiler Intrinsic Equivalent**

```
__m256 _mm256_maskload_ps(float const *a, __m256i mask)
void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)
__m256d _mm256_maskload_pd(double *a, __m256i mask);
void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);
__m128 _mm128_maskload_ps(float const *a, __m128i mask)
void _mm128_maskstore_ps(float *a, __m128i mask, __m128 b)
__m128d _mm128_maskload_pd(double *a, __m128i mask);
void _mm128_maskstore_pd(double *a, __m128i mask, __m128d b);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 6 (No AC# reported for any mask bit combinations);  
 additionally

#UD                      If VEX.W = 1.

## VPBLEND — Blend Packed Dwords

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F3A.W0 02 /r ib VPBLEND <i>xmm1, xmm2, xmm3/m128, imm8</i>	RVM1	V/V	AVX2	Select dwords from <i>xmm2</i> and <i>xmm3/m128</i> from mask specified in <i>imm8</i> and store the values into <i>xmm1</i> .
VEX.NDS.256.66.0F3A.W0 02 /r ib VPBLEND <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVM1	V/V	AVX2	Select dwords from <i>ymm2</i> and <i>ymm3/m256</i> from mask specified in <i>imm8</i> and store the values into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM1	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

### Description

Dword elements from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### VPBLEND (VEX.256 encoded version)

```

IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
ELSE DEST[127:96] ← SRC1[127:96]
IF (imm8[4] == 1) THEN DEST[159:128] ← SRC2[159:128]
ELSE DEST[159:128] ← SRC1[159:128]
IF (imm8[5] == 1) THEN DEST[191:160] ← SRC2[191:160]
ELSE DEST[191:160] ← SRC1[191:160]
IF (imm8[6] == 1) THEN DEST[223:192] ← SRC2[223:192]
ELSE DEST[223:192] ← SRC1[223:192]
IF (imm8[7] == 1) THEN DEST[255:224] ← SRC2[255:224]
ELSE DEST[255:224] ← SRC1[255:224]

```

**VPBLEND (VEX.128 encoded version)**

```

IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
ELSE DEST[127:96] ← SRC1[127:96]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPBLEND: `__m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)`

VPBLEND: `__m256i _mm256_blend_epi32 (__m256i v1, __m256i v2, const int mask)`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4; additionally

#UD If VEX.W = 1.

## VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 66 /r VPBLENDMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W0 66 /r VPBLENDMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W0 66 /r VPBLENDMB zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW	Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask.
EVEX.NDS.128.66.0F38.W1 66 /r VPBLENDMW xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W1 66 /r VPBLENDMW ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W1 66 /r VPBLENDMW zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW	Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).



**Operation****VPBLENDMB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SRC2[i+7:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN DEST[i+7:i] ← SRC1[i+7:i]
    ELSE                             ; zeroing-masking
      DEST[i+7:i] ← 0
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0;

```

**VPBLENDMW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC2[i+15:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN DEST[i+15:i] ← SRC1[i+15:i]
    ELSE                             ; zeroing-masking
      DEST[i+15:i] ← 0
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPBLENDMB __m512i __mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i __mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i __mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);
VPBLENDMW __m512i __mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);
VPBLENDMW __m256i __mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);
VPBLENDMW __m128i __mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 64 /r VPBLENDMD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512F	Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W0 64 /r VPBLENDMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512F	Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W0 64 /r VPBLENDMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F	Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask.
EVEX.NDS.128.66.0F38.W1 64 /r VPBLENDMQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512F	Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W1 64 /r VPBLENDMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL AVX512F	Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W1 64 /r VPBLENDMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F	Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

**Operation****VPBLENDMD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no controlmask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN DEST[i+31:i] ← SRC1[i+31:i]

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0;

**VPBLENDMD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no controlmask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC2[31:0]

ELSE

DEST[i+31:i] ← SRC2[i+31:i]

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN DEST[i+31:i] ← SRC1[i+31:i]

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```
VPBLENDMD __m512i _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);  
VPBLENDMD __m256i _mm256_mask_blend_epi32(__mmask8 m, __m256i a, __m256i b);  
VPBLENDMD __m128i _mm_mask_blend_epi32(__mmask8 m, __m128i a, __m128i b);  
VPBLENDMQ __m512i _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);  
VPBLENDMQ __m256i _mm256_mask_blend_epi64(__mmask8 m, __m256i a, __m256i b);  
VPBLENDMQ __m128i _mm_mask_blend_epi64(__mmask8 m, __m128i a, __m128i b);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPBROADCASTB/W/D/Q—Load with Broadcast Integer Data from General Purpose Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 7A /r VPBROADCASTB xmm1 {k1}{z}, reg	A	V/V	AVX512VL AVX512BW	Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7A /r VPBROADCASTB ymm1 {k1}{z}, reg	A	V/V	AVX512VL AVX512BW	Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7A /r VPBROADCASTB zmm1 {k1}{z}, reg	A	V/V	AVX512BW	Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7B /r VPBROADCASTW xmm1 {k1}{z}, reg	A	V/V	AVX512VL AVX512BW	Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7B /r VPBROADCASTW ymm1 {k1}{z}, reg	A	V/V	AVX512VL AVX512BW	Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7B /r VPBROADCASTW zmm1 {k1}{z}, reg	A	V/V	AVX512BW	Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7C /r VPBROADCASTD xmm1 {k1}{z}, r32	A	V/V	AVX512VL AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 {k1}{z}, r32	A	V/V	AVX512VL AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32	A	V/V	AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 {k1}{z}, r64	A	V/N.E. <sup>1</sup>	AVX512VL AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 {k1}{z}, r64	A	V/N.E. <sup>1</sup>	AVX512VL AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64	A	V/N.E. <sup>1</sup>	AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 512-bit destination subject to writemask k1.

### NOTES:

1. EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPBROADCASTB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← SRC[7:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPBROADCASTW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← SRC[15:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPBROADCASTD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[31:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPBROADCASTQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPBROADCASTB __m512i __mm512_mask_set1_epi8(__m512i s, __mmask64 k, int a);
VPBROADCASTB __m512i __mm512_maskz_set1_epi8(__mmask64 k, int a);
VPBROADCASTB __m256i __mm256_mask_set1_epi8(__m256i s, __mmask32 k, int a);
VPBROADCASTB __m256i __mm256_maskz_set1_epi8(__mmask32 k, int a);
VPBROADCASTB __m128i __mm128_mask_set1_epi8(__m128i s, __mmask16 k, int a);
VPBROADCASTB __m128i __mm128_maskz_set1_epi8(__mmask16 k, int a);
VPBROADCASTD __m512i __mm512_mask_set1_epi32(__m512i s, __mmask16 k, int a);
VPBROADCASTD __m512i __mm512_maskz_set1_epi32(__mmask16 k, int a);
VPBROADCASTD __m256i __mm256_mask_set1_epi32(__m256i s, __mmask8 k, int a);
VPBROADCASTD __m256i __mm256_maskz_set1_epi32(__mmask8 k, int a);
VPBROADCASTD __m128i __mm128_mask_set1_epi32(__m128i s, __mmask8 k, int a);
VPBROADCASTD __m128i __mm128_maskz_set1_epi32(__mmask8 k, int a);
VPBROADCASTQ __m512i __mm512_mask_set1_epi64(__m512i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m512i __mm512_maskz_set1_epi64(__mmask8 k, __int64 a);
VPBROADCASTQ __m256i __mm256_mask_set1_epi64(__m256i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m256i __mm256_maskz_set1_epi64(__mmask8 k, __int64 a);
VPBROADCASTQ __m128i __mm128_mask_set1_epi64(__m128i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m128i __mm128_maskz_set1_epi64(__mmask8 k, __int64 a);
VPBROADCASTW __m512i __mm512_mask_set1_epi16(__m512i s, __mmask32 k, int a);
VPBROADCASTW __m512i __mm512_maskz_set1_epi16(__mmask32 k, int a);
VPBROADCASTW __m256i __mm256_mask_set1_epi16(__m256i s, __mmask16 k, int a);
VPBROADCASTW __m256i __mm256_maskz_set1_epi16(__mmask16 k, int a);
VPBROADCASTW __m128i __mm128_mask_set1_epi16(__m128i s, __mmask8 k, int a);
VPBROADCASTW __m128i __mm128_maskz_set1_epi16(__mmask8 k, int a);

```

**Exceptions**

EVEX-encoded instructions, see Exceptions Type E7NM.

#UD If EVEX.vvvv != 1111B.

## VPBROADCAST—Load Integer and Broadcast

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to sixteen locations in xmm1.
VEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in ymm1.
EVEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1{k1}{z}, xmm2/m8	B	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1{k1}{z}, xmm2/m8	B	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 78 /r VPBROADCASTB zmm1{k1}{z}, xmm2/m8	B	V/V	AVX512BW	Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to eight locations in xmm1.
VEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in ymm1.
EVEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1{k1}{z}, xmm2/m16	B	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1{k1}{z}, xmm2/m16	B	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 79 /r VPBROADCASTW zmm1{k1}{z}, xmm2/m16	B	V/V	AVX512BW	Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in ymm1.
EVEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 58 /r VPBROADCASTD zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F	Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in source operand to two locations in xmm1.
VEX.256.66.0F38.W0 59 /r VPBROADCASTQ ymm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in source operand to four locations in ymm1.
EVEX.128.66.0F38.W1 59 /r VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 59 /r VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W1 59 /r VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F	Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 59 /r VPBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.



Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W0 59 /r VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 59 /r VBROADCASTI32x2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ	Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 ymm1, m128	A	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.
EVEX.256.66.0F38.W0 5A /r VBROADCASTI32X4 ymm1 {k1}{z}, m128	D	V/V	AVX512VL AVX512F	Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 5A /r VBROADCASTI32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F	Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 5A /r VBROADCASTI64X2 ymm1 {k1}{z}, m128	C	V/V	AVX512VL AVX512DQ	Broadcast 128 bits of 2 quadword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 5A /r VBROADCASTI64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ	Broadcast 128 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 5B /r VBROADCASTI32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ	Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 5B /r VBROADCASTI64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F	Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register.

VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VPBROADCASTI32X4 and VPBROADCASTI64X4 are reserved and will #UD.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.  
If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

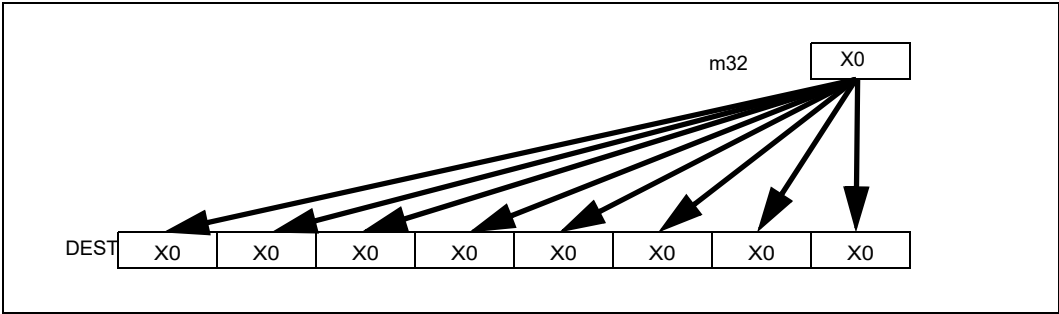


Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

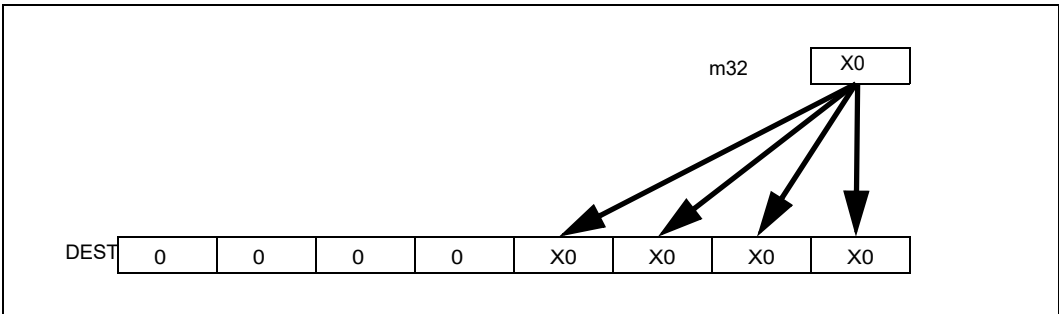


Figure 5-17. VPBROADCASTD Operation (128-bit version)

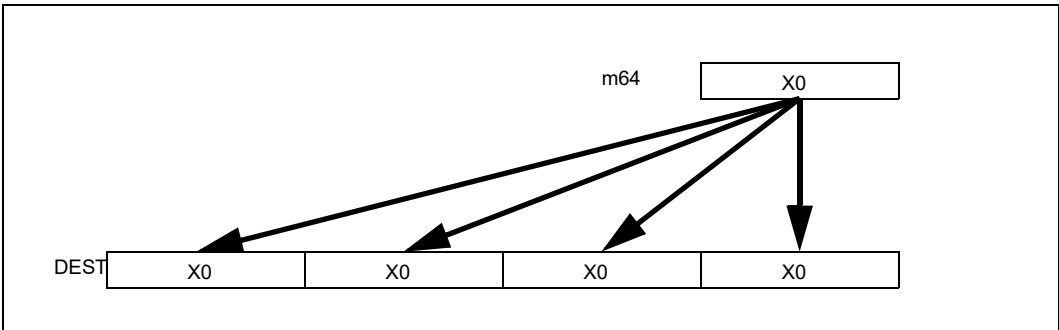


Figure 5-18. VPBROADCASTQ Operation (256-bit version)

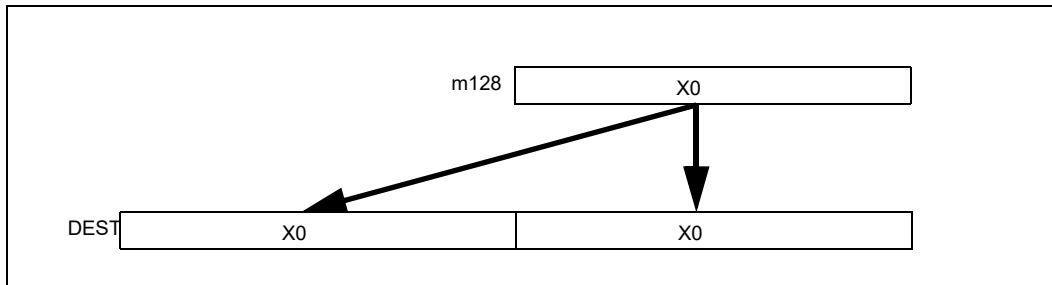


Figure 5-19. VBROADCASTI128 Operation (256-bit version)

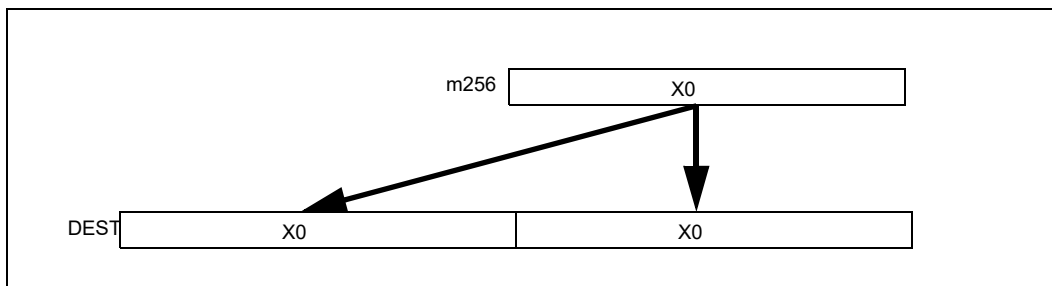


Figure 5-20. VBROADCASTI256 Operation (512-bit version)

**Operation****VPBROADCASTB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← SRC[7:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPBROADCASTW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← SRC[15:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPBROADCASTD (128 bit version)**

temp ← SRC[31:0]

DEST[31:0] ← temp

DEST[63:32] ← temp

DEST[95:64] ← temp

DEST[127:96] ← temp

DEST[MAXVL-1:128] ← 0

**VPBROADCASTD (VEX.256 encoded version)**

temp ← SRC[31:0]

DEST[31:0] ← temp

DEST[63:32] ← temp

DEST[95:64] ← temp

DEST[127:96] ← temp

DEST[159:128] ← temp

DEST[191:160] ← temp

DEST[223:192] ← temp

DEST[255:224] ← temp

DEST[MAXVL-1:256] ← 0

**VPBROADCASTD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[31:0]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPBROADCASTQ (VEX.256 encoded version)**

```

temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAXVL-1:256] ← 0

```

**VPBROADCASTQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VBROADCASTI32x2 (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    n ← (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VBROADCASTI128 (VEX.256 encoded version)**

```

temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAXVL-1:256] ← 0

```

**VBROADCASTI32X4 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

n ← (j modulo 4) \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[n+31:n]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VBROADCASTI64X2 (EVEX encoded versions)**

(KL, VL) = (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

n ← (j modulo 2) \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← SRC[n+63:n]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] = 0

FI

FI;

ENDFOR;

**VBROADCASTI32X8 (EVEX.U1.512 encoded version)**

FOR j ← 0 TO 15

i ← j \* 32

n ← (j modulo 8) \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+31:i] ← SRC[n+31:n]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VBROADCASTI64X4 (EVEX.512 encoded version)**

```

FOR j ← 0 TO 7
  i ← j * 64
  n ← (j modulo 4) * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[n+63:n]
  ELSE
    IF *merging-masking*                ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                                ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPBROADCASTB __m512i __mm512_broadcastb_epi8(__m128i a);
VPBROADCASTB __m512i __mm512_mask_broadcastb_epi8(__m512i s, __mmask64 k, __m128i a);
VPBROADCASTB __m512i __mm512_maskz_broadcastb_epi8(__mmask64 k, __m128i a);
VPBROADCASTB __m256i __mm256_broadcastb_epi8(__m128i a);
VPBROADCASTB __m256i __mm256_mask_broadcastb_epi8(__m256i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i __mm256_maskz_broadcastb_epi8(__mmask32 k, __m128i a);
VPBROADCASTB __m128i __mm_mask_broadcastb_epi8(__m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i __mm_maskz_broadcastb_epi8(__mmask16 k, __m128i a);
VPBROADCASTB __m128i __mm_broadcastb_epi8(__m128i a);
VPBROADCASTD __m512i __mm512_broadcastd_epi32(__m128i a);
VPBROADCASTD __m512i __mm512_mask_broadcastd_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCASTD __m512i __mm512_maskz_broadcastd_epi32(__mmask16 k, __m128i a);
VPBROADCASTD __m256i __mm256_broadcastd_epi32(__m128i a);
VPBROADCASTD __m256i __mm256_mask_broadcastd_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTD __m256i __mm256_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCASTD __m128i __mm_broadcastd_epi32(__m128i a);
VPBROADCASTD __m128i __mm_mask_broadcastd_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTD __m128i __mm_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCASTQ __m512i __mm512_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m512i __mm512_mask_broadcastq_epi64(__m512i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m512i __mm512_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTQ __m256i __mm256_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m256i __mm256_mask_broadcastq_epi64(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m256i __mm256_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTQ __m128i __mm_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m128i __mm_mask_broadcastq_epi64(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m128i __mm_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTW __m512i __mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i __mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCASTW __m512i __mm512_maskz_broadcastw_epi16(__mmask32 k, __m128i a);
VPBROADCASTW __m256i __mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i __mm256_mask_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCASTW __m256i __mm256_maskz_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCASTW __m128i __mm_broadcastw_epi16(__m128i a);
VPBROADCASTW __m128i __mm_mask_broadcastw_epi16(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTW __m128i __mm_maskz_broadcastw_epi16(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m512i __mm512_broadcast_i32x2(__m128i a);

```

VBROADCASTI32x2 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x2(\_\_m512i s, \_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x2(\_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m256i \_\_mm256\_broadcast\_i32x2(\_\_m128i a);  
 VBROADCASTI32x2 \_\_m256i \_\_mm256\_mask\_broadcast\_i32x2(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m256i \_\_mm256\_maskz\_broadcast\_i32x2(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m128i \_\_mm\_broadcastq\_i32x2(\_\_m128i a);  
 VBROADCASTI32x2 \_\_m128i \_\_mm\_mask\_broadcastq\_i32x2(\_\_m128i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m128i \_\_mm\_maskz\_broadcastq\_i32x2(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m512i \_\_mm512\_broadcast\_i32x4(\_\_m128i a);  
 VBROADCASTI32x4 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x4(\_\_m512i s, \_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x4(\_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_broadcast\_i32x4(\_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_mask\_broadcast\_i32x4(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_maskz\_broadcast\_i32x4(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_broadcast\_i32x8(\_\_m256i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x8(\_\_m512i s, \_\_mmask16 k, \_\_m256i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x8(\_\_mmask16 k, \_\_m256i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_broadcast\_i64x2(\_\_m128i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_mask\_broadcast\_i64x2(\_\_m512i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_maskz\_broadcast\_i64x2(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_broadcast\_i64x2(\_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_mask\_broadcast\_i64x2(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_maskz\_broadcast\_i64x2(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_broadcast\_i64x4(\_\_m256i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_mask\_broadcast\_i64x4(\_\_m512i s, \_\_mmask8 k, \_\_m256i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_maskz\_broadcast\_i64x4(\_\_mmask8 k, \_\_m256i a);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, syntax with reg/mem operand, see Exceptions Type E6.

#UD                      If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.  
                             If EVEX.L'L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.  
                             If EVEX.L'L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.



## VPBROADCASTM—Broadcast Mask to Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W1 2A /r VPBROADCASTMB2Q xmm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low byte value in k1 to two locations in xmm1.
EVEX.256.F3.0F38.W1 2A /r VPBROADCASTMB2Q ymm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low byte value in k1 to four locations in ymm1.
EVEX.512.F3.0F38.W1 2A /r VPBROADCASTMB2Q zmm1, k1	RM	V/V	AVX512CD	Broadcast low byte value in k1 to eight locations in zmm1.
EVEX.128.F3.0F38.W0 3A /r VPBROADCASTMW2D xmm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low word value in k1 to four locations in xmm1.
EVEX.256.F3.0F38.W0 3A /r VPBROADCASTMW2D ymm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low word value in k1 to eight locations in ymm1.
EVEX.512.F3.0F38.W0 3A /r VPBROADCASTMW2D zmm1, k1	RM	V/V	AVX512CD	Broadcast low word value in k1 to sixteen locations in zmm1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128).

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VPBROADCASTMB2Q

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j\*64

    DEST[i+63:i] ← ZeroExtend(SRC[7:0])

ENDFOR

DEST[MAXVL-1:VL] ← 0

#### VPBROADCASTMW2D

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j\*32

    DEST[i+31:i] ← ZeroExtend(SRC[15:0])

ENDFOR

DEST[MAXVL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q \_\_m512i \_\_mm512\_broadcastmb\_epi64(\_\_mmask8);  
VPBROADCASTMW2D \_\_m512i \_\_mm512\_broadcastmw\_epi32(\_\_mmask16);  
VPBROADCASTMB2Q \_\_m256i \_\_mm256\_broadcastmb\_epi64(\_\_mmask8);  
VPBROADCASTMW2D \_\_m256i \_\_mm256\_broadcastmw\_epi32(\_\_mmask8);  
VPBROADCASTMB2Q \_\_m128i \_\_mm\_broadcastmb\_epi64(\_\_mmask8);  
VPBROADCASTMW2D \_\_m128i \_\_mm\_broadcastmw\_epi32(\_\_mmask8);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF.

## VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 3F /r ib  VPCMPB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL AVX512BW	Compare packed signed byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 3F /r ib  VPCMPB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL AVX512BW	Compare packed signed byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW	Compare packed signed byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.128.66.0F3A.W0 3E /r ib  VPCMPUB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL AVX512BW	Compare packed unsigned byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 3E /r ib  VPCMPUB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL AVX512BW	Compare packed unsigned byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW	Compare packed unsigned byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-17.

Table 5-17. Pseudo-Op and VPCMP\* Implementation

Pseudo-Op	PCMPM Implementation
VPCMPEQ* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 0</i>
VPCMPLT* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 1</i>
VPCMPLE* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 2</i>
VPCMPNEQ* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 4</i>
VPPCMPNLT* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 5</i>
VPCMPNLE* <i>reg1, reg2, reg3</i>	VPCMP* <i>reg1, reg2, reg3, 6</i>

Operation

```
CASE (COMPARISON PREDICATE) OF
  0: OP ← EQ;
  1: OP ← LT;
  2: OP ← LE;
  3: OP ← FALSE;
  4: OP ← NEQ;
  5: OP ← NLT;
  6: OP ← NLE;
  7: OP ← TRUE;
ESAC;

VPCMPB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k2[j] OR *no writemask*
    THEN
      CMP ← SRC1[i+7:i] OP SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE   DEST[j] = 0           ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**VPCMPUB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF k2[j] OR \*no writemask\*

THEN

CMP ← SRC1[i+7:i] OP SRC2[i+7:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPCMPB __mmask64 __mm512_cmp_epi8_mask( __m512i a, __m512i b, int cmp);
VPCMPB __mmask64 __mm512_mask_cmp_epi8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPB __mmask32 __mm256_cmp_epi8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __mmask32 __mm256_mask_cmp_epi8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPB __mmask16 __mm128_cmp_epi8_mask( __m128i a, __m128i b, int cmp);
VPCMPB __mmask16 __mm128_mask_cmp_epi8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPB __mmask64 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m512i a, __m512i b);
VPCMPB __mmask64 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32 __mm256_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m256i a, __m256i b);
VPCMPB __mmask32 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16 __mm128_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m128i a, __m128i b);
VPCMPB __mmask16 __mm128_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask16 m, __m128i a, __m128i b);
VPCMPUB __mmask64 __mm512_cmp_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 __mm512_mask_cmp_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 __mm256_cmp_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 __mm256_mask_cmp_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 __mm128_cmp_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 __mm128_mask_cmp_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPUB __mmask64 __mm512_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 __mm256_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 __mm128_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 __mm128_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask( __mmask16 m, __m128i a, __m128i b, int cmp);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed signed doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed signed doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.
EVEX.NDS.128.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed unsigned doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed unsigned doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPD/VPCMPUD performs a comparison between pairs of signed/unsigned doubleword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register k1. Up to 16/8/4 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-17.

**Operation**

CASE (COMPARISON PREDICATE) OF

```

0: OP ← EQ;
1: OP ← LT;
2: OP ← LE;
3: OP ← FALSE;
4: OP ← NEQ;
5: OP ← NLT;
6: OP ← NLE;
7: OP ← TRUE;

```

ESAC;

**VPCMPD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k2[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

        ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

      FI;

      IF CMP = TRUE

        THEN DEST[j] ← 1;

        ELSE DEST[j] ← 0; FI;

    ELSE     DEST[j] ← 0                   ; zeroing-masking onlyFI;

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k2[j] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

        ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

      FI;

      IF CMP = TRUE

        THEN DEST[j] ← 1;

        ELSE DEST[j] ← 0; FI;

    ELSE     DEST[j] ← 0                   ; zeroing-masking onlyFI;

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPCMPD __mmask16 _mm512_cmp_epi32_mask( __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_mask_cmp_epi32_mask( __mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m512i a, __m512i b);
VPCMPD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_cmp_epu32_mask( __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_mask_cmp_epu32_mask( __mmask16 k, __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __mmask16 k, __m512i a, __m512i b);
VPCMPD __mmask8 _mm256_cmp_epi32_mask( __m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_mask_cmp_epi32_mask( __mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m256i a, __m256i b);
VPCMPD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __mmask8 k, __m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_cmp_epu32_mask( __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_mask_cmp_epu32_mask( __mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 _mm_cmp_epi32_mask( __m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_mask_cmp_epi32_mask( __mmask8 k, __m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m128i a, __m128i b);
VPCMPD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __mmask8 k, __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_cmp_epu32_mask( __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_mask_cmp_epu32_mask( __mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __mmask8 k, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.



## VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed signed quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed signed quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.128.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed unsigned quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed unsigned quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-17.

**Operation**

CASE (COMPARISON PREDICATE) OF

0: OP ← EQ;  
 1: OP ← LT;  
 2: OP ← LE;  
 3: OP ← FALSE;  
 4: OP ← NEQ;  
 5: OP ← NLT;  
 6: OP ← NLE;  
 7: OP ← TRUE;

ESAC;

**VPCMPQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];

ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];

ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPCMPQ __mmask8_mm512_cmp_epi64_mask( __m512i a, __m512i b, int imm);
VPCMPQ __mmask8_mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8_mm512_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m512i a, __m512i b);
VPCMPQ __mmask8_mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPUQ __mmask8_mm512_cmp_epu64_mask( __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8_mm512_mask_cmp_epu64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8_mm512_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m512i a, __m512i b);
VPCMPUQ __mmask8_mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPQ __mmask8_mm256_cmp_epi64_mask( __m256i a, __m256i b, int imm);
VPCMPQ __mmask8_mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8_mm256_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m256i a, __m256i b);
VPCMPQ __mmask8_mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUQ __mmask8_mm256_cmp_epu64_mask( __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8_mm256_mask_cmp_epu64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8_mm256_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m256i a, __m256i b);
VPCMPUQ __mmask8_mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPQ __mmask8_mm_cmp_epi64_mask( __m128i a, __m128i b, int imm);
VPCMPQ __mmask8_mm_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8_mm_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m128i a, __m128i b);
VPCMPQ __mmask8_mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUQ __mmask8_mm_cmp_epu64_mask( __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8_mm_mask_cmp_epu64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8_mm_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m128i a, __m128i b);
VPCMPUQ __mmask8_mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m128i a, __m128i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 3F /r ib  VPCMPW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 3F /r ib  VPCMPW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW	Compare packed signed word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.128.66.0F3A.W1 3E /r ib  VPCMPUW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 3E /r ib  VPCMPUW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
VPCMPUW k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW	Compare packed unsigned word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-17.

**Operation**

CASE (COMPARISON PREDICATE) OF

0: OP  $\leftarrow$  EQ;  
 1: OP  $\leftarrow$  LT;  
 2: OP  $\leftarrow$  LE;  
 3: OP  $\leftarrow$  FALSE;  
 4: OP  $\leftarrow$  NEQ;  
 5: OP  $\leftarrow$  NLT;  
 6: OP  $\leftarrow$  NLE;  
 7: OP  $\leftarrow$  TRUE;

ESAC;

**VPCMPW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j  $\leftarrow$  0 TO KL-1

i  $\leftarrow$  j \* 16

IF k2[j] OR \*no writemask\*

THEN

ICMP  $\leftarrow$  SRC1[i+15:i] OP SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j]  $\leftarrow$  1;

ELSE DEST[j]  $\leftarrow$  0; FI;

ELSE DEST[j] = 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL]  $\leftarrow$  0

**VPCMPUW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j  $\leftarrow$  0 TO KL-1

i  $\leftarrow$  j \* 16

IF k2[j] OR \*no writemask\*

THEN

CMP  $\leftarrow$  SRC1[i+15:i] OP SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j]  $\leftarrow$  1;

ELSE DEST[j]  $\leftarrow$  0; FI;

ELSE DEST[j] = 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPCMPW __mmask32 __mm512_cmp_epi16_mask( __m512i a, __m512i b, int cmp);
VPCMPW __mmask32 __mm512_mask_cmp_epi16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPW __mmask16 __mm256_cmp_epi16_mask( __m256i a, __m256i b, int cmp);
VPCMPW __mmask16 __mm256_mask_cmp_epi16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask8 __mm_cmp_epi16_mask( __m128i a, __m128i b, int cmp);
VPCMPW __mmask8 __mm_mask_cmp_epi16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m512i a, __m512i b);
VPCMPW __mmask32 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 __mm256_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m256i a, __m256i b);
VPCMPW __mmask16 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask8 __mm_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m128i a, __m128i b);
VPCMPW __mmask8 __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask8 m, __m128i a, __m128i b);
VPCMPUW __mmask32 __mm512_cmp_epu16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 __mm512_mask_cmp_epu16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 __mm256_cmp_epu16_mask( __m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 __mm256_mask_cmp_epu16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 __mm_cmp_epu16_mask( __m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 __mm_mask_cmp_epu16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPUW __mmask32 __mm512_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 __mm256_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 __mm_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8B /r VPCOMPRESSD xmm1/m128 {k1}{z}, xmm2	A	V/V	AVX512VL AVX512F	Compress packed doubleword integer values from xmm2 to xmm1/m128 using controlmask k1.
EVEX.256.66.0F38.W0 8B /r VPCOMPRESSD ymm1/m256 {k1}{z}, ymm2	A	V/V	AVX512VL AVX512F	Compress packed doubleword integer values from ymm2 to ymm1/m256 using controlmask k1.
EVEX.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F	Compress packed doubleword integer values from zmm2 to zmm1/m512 using controlmask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VPCOMPRESSD (EVEX encoded versions) store form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no controlmask\*

    THEN

      DEST[k+SIZE-1:k] ← SRC[i+31:i]

      k ← k + SIZE

  FI;

ENDFOR;

**VPCOMPRESSD (EVEX encoded versions) reg-reg form**

(KL, VL) = (4, 128), (8, 256), (16, 512)

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no controlmask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+31:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCOMPRESSD \_\_m512i \_\_mm512\_mask\_compress\_epi32(\_\_m512i s, \_\_mmask16 c, \_\_m512i a);

VPCOMPRESSD \_\_m512i \_\_mm512\_maskz\_compress\_epi32(\_\_mmask16 c, \_\_m512i a);

VPCOMPRESSD void \_\_mm512\_mask\_compressstoreu\_epi32(void \* a, \_\_mmask16 c, \_\_m512i s);

VPCOMPRESSD \_\_m256i \_\_mm256\_mask\_compress\_epi32(\_\_m256i s, \_\_mmask8 c, \_\_m256i a);

VPCOMPRESSD \_\_m256i \_\_mm256\_maskz\_compress\_epi32(\_\_mmask8 c, \_\_m256i a);

VPCOMPRESSD void \_\_mm256\_mask\_compressstoreu\_epi32(void \* a, \_\_mmask8 c, \_\_m256i s);

VPCOMPRESSD \_\_m128i \_\_mm\_mask\_compress\_epi32(\_\_m128i s, \_\_mmask8 c, \_\_m128i a);

VPCOMPRESSD \_\_m128i \_\_mm\_maskz\_compress\_epi32(\_\_mmask8 c, \_\_m128i a);

VPCOMPRESSD void \_\_mm\_mask\_compressstoreu\_epi32(void \* a, \_\_mmask8 c, \_\_m128i s);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.



## VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8B /r VPCOMPRESSQ xmm1/m128 {k1}{z}, xmm2	A	V/V	AVX512VL AVX512F	Compress packed quadword integer values from xmm2 to xmm1/m128 using controlmask k1.
EVEX.256.66.0F38.W1 8B /r VPCOMPRESSQ ymm1/m256 {k1}{z}, ymm2	A	V/V	AVX512VL AVX512F	Compress packed quadword integer values from ymm2 to ymm1/m256 using controlmask k1.
EVEX.512.66.0F38.W1 8B /r VPCOMPRESSQ zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F	Compress packed quadword integer values from zmm2 to zmm1/m512 using controlmask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VPCOMPRESSQ (EVEX encoded versions) store form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no controlmask\*

        THEN

            DEST[k+SIZE-1:k] ← SRC[i+63:i]

            k ← k + SIZE

    FI;

ENFOR

**VPCOMPRESSQ (EVEX encoded versions) reg-reg form**

(KL, VL) = (2, 128), (4, 256), (8, 512)

SIZE ← 64

k ← 0

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no controlmask\*

THEN

DEST[k+SIZE-1:k] ← SRC[i+63:i]

k ← k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] ← 0

FI

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPCOMPRESSQ \_\_m512i \_mm512\_mask\_compress\_epi64(\_\_m512i s, \_\_mmask8 c, \_\_m512i a);

VPCOMPRESSQ \_\_m512i \_mm512\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m512i a);

VPCOMPRESSQ void \_mm512\_mask\_compressstoreu\_epi64(void \* a, \_\_mmask8 c, \_\_m512i s);

VPCOMPRESSQ \_\_m256i \_mm256\_mask\_compress\_epi64(\_\_m256i s, \_\_mmask8 c, \_\_m256i a);

VPCOMPRESSQ \_\_m256i \_mm256\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m256i a);

VPCOMPRESSQ void \_mm256\_mask\_compressstoreu\_epi64(void \* a, \_\_mmask8 c, \_\_m256i s);

VPCOMPRESSQ \_\_m128i \_mm\_mask\_compress\_epi64(\_\_m128i s, \_\_mmask8 c, \_\_m128i a);

VPCOMPRESSQ \_\_m128i \_mm\_maskz\_compress\_epi64(\_\_mmask8 c, \_\_m128i a);

VPCOMPRESSQ void \_mm\_mask\_compressstoreu\_epi64(void \* a, \_\_mmask8 c, \_\_m128i s);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

## VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/ Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 C4 /r VPCONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 C4 /r VPCONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD	Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 C4 /r VPCONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 C4 /r VPCONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD	Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element's comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPCONFLICTD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j\*32

IF MaskBit(j) OR \*no writemask\* THEN

FOR k ← 0 TO j-1

m ← k\*32

IF ((SRC[i+31:i] = SRC[m+31:m])) THEN

DEST[i+k] ← 1

ELSE

DEST[i+k] ← 0

FI

ENDFOR

DEST[i+31:i+j] ← 0

ELSE

IF \*merging-masking\* THEN

\*DEST[i+31:i] remains unchanged\*

ELSE

DEST[i+31:i] ← 0

FI

FI

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPCONFLICTQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j\*64

IF MaskBit(j) OR \*no writemask\* THEN

FOR k ← 0 TO j-1

m ← k\*64

IF ((SRC[i+63:i] = SRC[m+63:m])) THEN

DEST[i+k] ← 1

ELSE

DEST[i+k] ← 0

FI

ENDFOR

DEST[i+63:i+j] ← 0

ELSE

IF \*merging-masking\* THEN

\*DEST[i+63:i] remains unchanged\*

ELSE

DEST[i+63:i] ← 0

FI

FI

ENDFOR

DEST[MAXVL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPCONFLICTD __m512i _mm512_conflict_epi32( __m512i a);
VPCONFLICTD __m512i _mm512_mask_conflict_epi32(__m512i s, __mmask16 m, __m512i a);
VPCONFLICTD __m512i _mm512_maskz_conflict_epi32(__mmask16 m, __m512i a);
VPCONFLICTQ __m512i _mm512_conflict_epi64( __m512i a);
VPCONFLICTQ __m512i _mm512_mask_conflict_epi64(__m512i s, __mmask8 m, __m512i a);
VPCONFLICTQ __m512i _mm512_maskz_conflict_epi64(__mmask8 m, __m512i a);
VPCONFLICTD __m256i _mm256_conflict_epi32( __m256i a);
VPCONFLICTD __m256i _mm256_mask_conflict_epi32(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTD __m256i _mm256_maskz_conflict_epi32(__mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_conflict_epi64( __m256i a);
VPCONFLICTQ __m256i _mm256_mask_conflict_epi64(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_maskz_conflict_epi64(__mmask8 m, __m256i a);
VPCONFLICTD __m128i _mm_conflict_epi32( __m128i a);
VPCONFLICTD __m128i _mm_mask_conflict_epi32(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTD __m128i _mm_maskz_conflict_epi32(__mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_conflict_epi64( __m128i a);
VPCONFLICTQ __m128i _mm_mask_conflict_epi64(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_maskz_conflict_epi64(__mmask8 m, __m128i a);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPERM2F128 — Permute Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 06 /r ib VPERM2F128 <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Permute 128-bit floating-point fields in <i>ymm2</i> and <i>ymm3/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

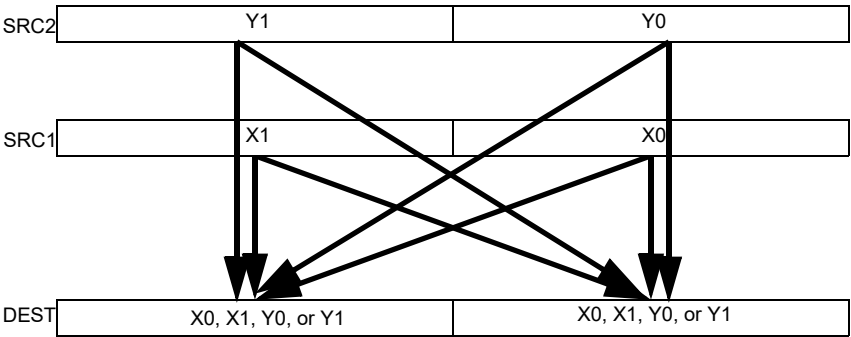


Figure 5-21. VPERM2F128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.

## Operation

### VPERM2F128

CASE IMM8[1:0] of

0:  $\text{DEST}[127:0] \leftarrow \text{SRC1}[127:0]$

1:  $\text{DEST}[127:0] \leftarrow \text{SRC1}[255:128]$

2:  $\text{DEST}[127:0] \leftarrow \text{SRC2}[127:0]$

3:  $\text{DEST}[127:0] \leftarrow \text{SRC2}[255:128]$

ESAC

CASE IMM8[5:4] of

0:  $\text{DEST}[255:128] \leftarrow \text{SRC1}[127:0]$

1:  $\text{DEST}[255:128] \leftarrow \text{SRC1}[255:128]$

2:  $\text{DEST}[255:128] \leftarrow \text{SRC2}[127:0]$

3:  $\text{DEST}[255:128] \leftarrow \text{SRC2}[255:128]$

ESAC

IF (imm8[3])

$\text{DEST}[127:0] \leftarrow 0$

FI

IF (imm8[7])

$\text{DEST}[\text{MAXVL}-1:128] \leftarrow 0$

FI

### Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128: `__m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)`

VPERM2F128: `__m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)`

VPERM2F128: `__m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 6; additionally

#UD                    If VEX.L = 0  
                          If VEX.W = 1.

VPERM2I128 – Permute Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 46 /r ib VPERM2I128 <i>ymm1</i> , <i>ymm2</i> , <i>ymm3/m256</i> , <i>imm8</i>	RVMI	V/V	AVX2	Permute 128-bit integer data in <i>ymm2</i> and <i>ymm3/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

Description

Permute 128 bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

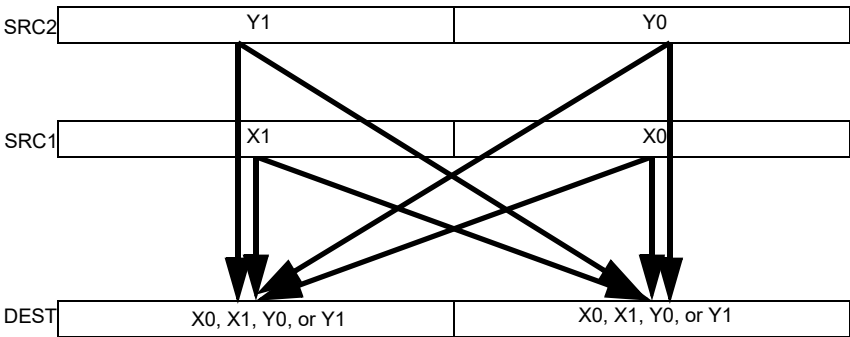


Figure 5-22. VPERM2I128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.



## Operation

### VPERM2I128

CASE IMM8[1:0] of

0:  $\text{DEST}[127:0] \leftarrow \text{SRC1}[127:0]$

1:  $\text{DEST}[127:0] \leftarrow \text{SRC1}[255:128]$

2:  $\text{DEST}[127:0] \leftarrow \text{SRC2}[127:0]$

3:  $\text{DEST}[127:0] \leftarrow \text{SRC2}[255:128]$

ESAC

CASE IMM8[5:4] of

0:  $\text{DEST}[255:128] \leftarrow \text{SRC1}[127:0]$

1:  $\text{DEST}[255:128] \leftarrow \text{SRC1}[255:128]$

2:  $\text{DEST}[255:128] \leftarrow \text{SRC2}[127:0]$

3:  $\text{DEST}[255:128] \leftarrow \text{SRC2}[255:128]$

ESAC

IF (imm8[3])

$\text{DEST}[127:0] \leftarrow 0$

FI

IF (imm8[7])

$\text{DEST}[255:128] \leftarrow 0$

FI

### Intel C/C++ Compiler Intrinsic Equivalent

VPERM2I128: `__m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, int control)`

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

## VPERMB—Permute Packed Bytes Elements

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 8D /r VPERMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512_VBMI	Permute bytes in xmm3/m128 using byte indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 8D /r VPERMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512_VBMI	Permute bytes in ymm3/m256 using byte indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 8D /r VPERMB zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI	Permute bytes in zmm3/m512 using byte indexes in zmm2 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Copies bytes from the second source operand (the third operand) to the destination operand (the first operand) according to the byte indices in the first source operand (the second operand). Note that this instruction permits a byte in the source operand to be copied to more than one location in the destination operand.

Only the low 6(EVEX.512)/5(EVEX.256)/4(EVEX.128) bits of each byte index is used to select the location of the source byte from the second source operand.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated at byte granularity by the writemask k1.

### Operation

#### VPERMB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

n ← 3;

ELSE IF VL = 256:

n ← 4;

ELSE IF VL = 512:

n ← 5;

FI;

FOR j ← 0 TO KL-1:

id ← SRC1[j\*8 + n : j\*8] ; // location of the source byte

IF k1[j] OR \*no writemask\* THEN

DEST[j\*8 + 7 : j\*8] ← SRC2[id\*8 + 7 : id\*8];

ELSE IF zeroing-masking THEN

DEST[j\*8 + 7 : j\*8] ← 0;

\*ELSE

DEST[j\*8 + 7 : j\*8] remains unchanged\*

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

### Intel C/C++ Compiler Intrinsic Equivalent

VPERMB \_\_m512i \_\_mm512\_permutexvar\_epi8(\_\_m512i idx, \_\_m512i a);

```

VPERMB __m512i __mm512_mask_permutexvar_epi8(__m512i s, __mmask64 k, __m512i idx, __m512i a);
VPERMB __m512i __mm512_maskz_permutexvar_epi8(__mmask64 k, __m512i idx, __m512i a);
VPERMB __m256i __mm256_permutexvar_epi8(__m256i idx, __m256i a);
VPERMB __m256i __mm256_mask_permutexvar_epi8(__m256i s, __mmask32 k, __m256i idx, __m256i a);
VPERMB __m256i __mm256_maskz_permutexvar_epi8(__mmask32 k, __m256i idx, __m256i a);
VPERMB __m128i __mm_permutexvar_epi8(__m128i idx, __m128i a);
VPERMB __m128i __mm_mask_permutexvar_epi8(__m128i s, __mmask16 k, __m128i idx, __m128i a);
VPERMB __m128i __mm_maskz_permutexvar_epi8(__mmask16 k, __m128i idx, __m128i a);

```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type E4NF.nb.

## VPERMD/VPERMW—Permute Packed Doublewords/Words Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 36 /r VPERMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 8D /r  VPERMW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Permute word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 8D /r  VPERMW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Permute word integers in ymm3/m256 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 8D /r VPERMW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
B	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Copies doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

**Operation****VPERMD (EVEX encoded versions)**

```

(KL, VL) = (8, 256), (16, 512)
IF VL = 256 THEN n ← 2; FI;
IF VL = 512 THEN n ← 3; FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    id ← 32*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC2[31:0];
                ELSE DEST[i+31:i] ← SRC2[id+31:id];
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPERMD (VEX.256 encoded version)**

```

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] ← 0

```

**VPERMW (EVEX encoded versions)**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128 THEN n ← 2; FI;
IF VL = 256 THEN n ← 3; FI;
IF VL = 512 THEN n ← 4; FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    id ← 16*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC2[id+15:id]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMD __m512i __mm512_permutexvar_epi32(__m512i idx, __m512i a);
VPERMD __m512i __mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i __mm512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i __mm256_permutexvar_epi32(__m256i idx, __m256i a);
VPERMD __m256i __mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i __mm256_maskz_permutexvar_epi32(__mmask8 k, __m256i idx, __m256i a);
VPERMW __m512i __mm512_permutexvar_epi16(__m512i idx, __m512i a);
VPERMW __m512i __mm512_mask_permutexvar_epi16(__m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i __mm512_maskz_permutexvar_epi16(__mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i __mm256_permutexvar_epi16(__m256i idx, __m256i a);
VPERMW __m256i __mm256_mask_permutexvar_epi16(__m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i __mm256_maskz_permutexvar_epi16(__mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i __mm_permutexvar_epi16(__m128i idx, __m128i a);
VPERMW __m128i __mm_mask_permutexvar_epi16(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i __mm_maskz_permutexvar_epi16(__mmask8 k, __m128i idx, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPERMD, see Exceptions Type E4NF.

EVEX-encoded VPERMW, see Exceptions Type E4NF.nb.

#UD                      If VEX.L = 0.  
                             If EVEX.L'L = 0 for VPERMD.

## VPERMI2B—Full Permute of Bytes from Two Tables Overwriting the Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W0 75 /r VPERMI2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512_VBMI	Permute bytes in xmm3/m128 and xmm2 using byte indexes in xmm1 and store the byte results in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 75 /r VPERMI2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512_VBMI	Permute bytes in ymm3/m256 and ymm2 using byte indexes in ymm1 and store the byte results in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 75 /r VPERMI2B zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI	Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Permutes byte values in the second operand (the first source operand) and the third operand (the second source operand) using the byte indices in the first operand (the destination operand) to select byte elements from the second or third source operands. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result. The third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the same tables can be reused in subsequent iterations, but the index elements are overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****VPERMI2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

id ← 3;

ELSE IF VL = 256:

id ← 4;

ELSE IF VL = 512:

id ← 5;

FI;

TMP\_DEST[VL-1:0] ← DEST[VL-1:0];

FOR j ← 0 TO KL-1

off ← 8\*SRC1[j\*8 + id:j\*8];

IF k1[j] OR \*no writemask\*:

DEST[j\*8 + 7:j\*8] ← TMP\_DEST[j\*8+id+1]: SRC2[off+7:off] : SRC1[off+7:off];

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7:j\*8] ← 0;

\*ELSE

DEST[j\*8 + 7:j\*8] remains unchanged\*

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMI2B \_\_m512i \_\_mm512\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI2B \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_mmask64 k, \_\_m512i b);

VPERMI2B \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI2B \_\_m256i \_\_mm256\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2B \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_mmask32 k, \_\_m256i b);

VPERMI2B \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2B \_\_m128i \_\_mm\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMI2B \_\_m128i \_\_mm\_mask2\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_mmask16 k, \_\_m128i b);

VPERMI2B \_\_m128i \_\_mm\_maskz\_permutex2var\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.



## VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 75 /r VPERMI2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 75 /r VPERMI2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 75 /r VPERMI2W zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW	Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 76 /r VPERMI2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 76 /r VPERMI2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 76 /r VPERMI2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W1 76 /r VPERMI2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 76 /r VPERMI2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 76 /r VPERMI2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 77 /r VPERMI2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 77 /r VPERMI2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 77 /r VPERMI2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 77 /r VPERMI2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 77 /r VPERMI2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Permutes 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table\_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table\_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same table can be reused for example for a second iteration, while the index elements are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****VPERMI2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

id ← 2

FI;

IF VL = 256

id ← 3

FI;

IF VL = 512

id ← 4

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 16

off ← 16 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

DEST[i+15:i] = TMP\_DEST[i+id+1] ? SRC2[off+15:off]

: SRC1[off+15:off]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPERMI2D/VPERMI2PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

id ← 1

FI;

IF VL = 256

id ← 2

FI;

IF VL = 512

id ← 3

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 32

off ← 32 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← TMP\_DEST[i+id+1] ? SRC2[31:0]

: SRC1[off+31:off]

ELSE

DEST[i+31:i] ← TMP\_DEST[i+id+1] ? SRC2[off+31:off]

: SRC1[off+31:off]

```

        FI
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[j+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[j+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPERMI2Q/VPERMI2PD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8 512)

IF VL = 128

id ← 0

FI;

IF VL = 256

id ← 1

FI;

IF VL = 512

id ← 2

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 64

off ← 64 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

```

        DEST[i+63:i] ← TMP_DEST[j+id+1] ? SRC2[63:0]
        : SRC1[off+63:off]

```

ELSE

```

        DEST[i+63:i] ← TMP_DEST[j+id+1] ? SRC2[off+63:off]
        : SRC1[off+63:off]

```

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[j+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[j+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

## Intel C/C++ Compiler Intrinsic Equivalent

```

VPERMI2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMI2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMI __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMI2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMI2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMI2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMI2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMI2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMI2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMI2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMI2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERMI2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMI2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMI2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMI2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMI2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMI2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMI2Q __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2Q __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2Q __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2Q __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);

```

VPERMI2W \_\_m512i \_\_mm512\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2W \_\_m512i \_\_mm512\_mask\_permutex2var\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m512i idx, \_\_m512i b);  
 VPERMI2W \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_mmask32 k, \_\_m512i b);  
 VPERMI2W \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2W \_\_m256i \_\_mm256\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2W \_\_m256i \_\_mm256\_mask\_permutex2var\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m256i idx, \_\_m256i b);  
 VPERMI2W \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_mmask16 k, \_\_m256i b);  
 VPERMI2W \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2W \_\_m128i \_\_mm\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2W \_\_m128i \_\_mm\_mask\_permutex2var\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMI2W \_\_m128i \_\_mm\_mask2\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMI2W \_\_m128i \_\_mm\_maskz\_permutex2var\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VPERMI2D/Q/PS/PD: See Exceptions Type E4NF.

VPERMI2W: See Exceptions Type E4NF.nb.

## VPERMILPD—Permute In-Lane of Pairs of Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute double-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.NDS.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute double-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
EVEX.NDS.128.66.0F38.W1 0D /r VPERMILPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 0D /r VPERMILPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 0D /r VPERMILPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Permute double-precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute double-precision floating-point values in xmm2/m128 using controls from imm8.
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute double-precision floating-point values in ymm2/m256 using controls from imm8.
EVEX.128.66.0F3A.W1 05 /r ib VPERMILPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	D	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W1 05 /r ib VPERMILPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	D	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W1 05 /r ib VPERMILPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	D	V/V	AVX512F	Permute double-precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

(variable control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permuted results are stored in the destination operand (first operand).

The control bits are located at bit 0 of each quadword element (see Figure 5-24). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.

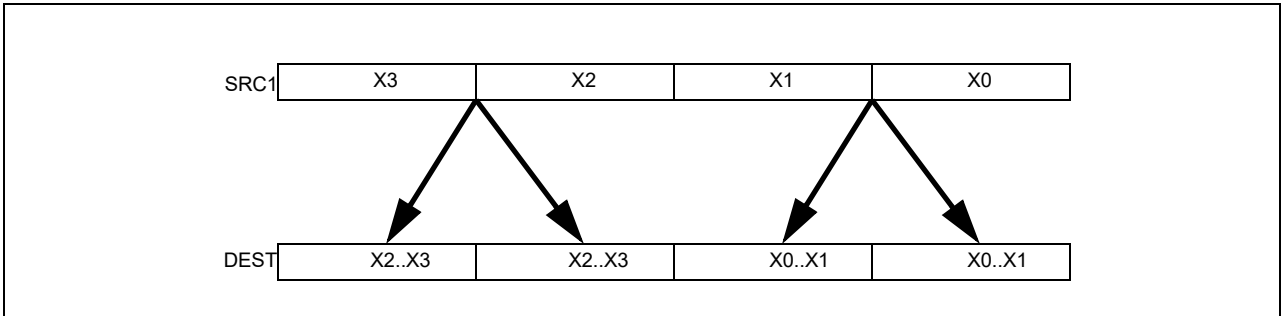


Figure 5-23. VPERMILPD Operation

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

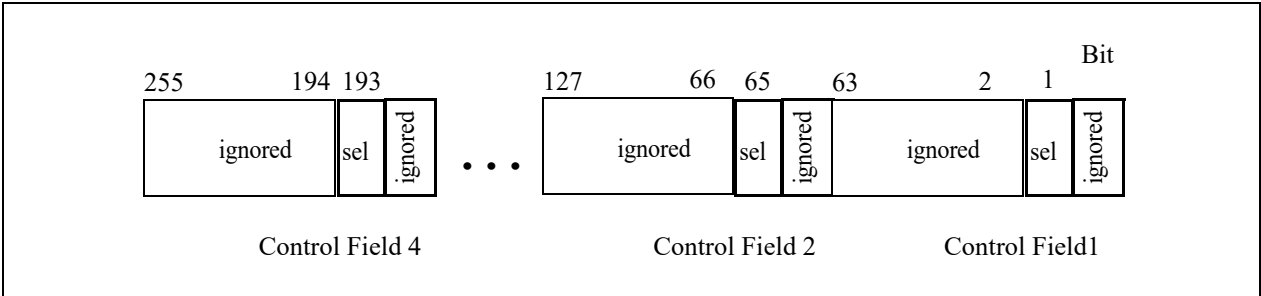


Figure 5-24. VPERMILPD Shuffle Control

(immediate control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.



**Operation****VPERMILPD (EVEX immediate versions)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN TMP\_SRC1[i+63:i] ← SRC1[63:0];

ELSE TMP\_SRC1[i+63:i] ← SRC1[i+63:i];

FI;

ENDFOR;

IF (imm8[0] = 0) THEN TMP\_DEST[63:0] ← SRC1[63:0]; FI;

IF (imm8[0] = 1) THEN TMP\_DEST[63:0] ← TMP\_SRC1[127:64]; FI;

IF (imm8[1] = 0) THEN TMP\_DEST[127:64] ← TMP\_SRC1[63:0]; FI;

IF (imm8[1] = 1) THEN TMP\_DEST[127:64] ← TMP\_SRC1[127:64]; FI;

IF VL &gt;= 256

IF (imm8[2] = 0) THEN TMP\_DEST[191:128] ← TMP\_SRC1[191:128]; FI;

IF (imm8[2] = 1) THEN TMP\_DEST[191:128] ← TMP\_SRC1[255:192]; FI;

IF (imm8[3] = 0) THEN TMP\_DEST[255:192] ← TMP\_SRC1[191:128]; FI;

IF (imm8[3] = 1) THEN TMP\_DEST[255:192] ← TMP\_SRC1[255:192]; FI;

FI;

IF VL &gt;= 512

IF (imm8[4] = 0) THEN TMP\_DEST[319:256] ← TMP\_SRC1[319:256]; FI;

IF (imm8[4] = 1) THEN TMP\_DEST[319:256] ← TMP\_SRC1[383:320]; FI;

IF (imm8[5] = 0) THEN TMP\_DEST[383:320] ← TMP\_SRC1[319:256]; FI;

IF (imm8[5] = 1) THEN TMP\_DEST[383:320] ← TMP\_SRC1[383:320]; FI;

IF (imm8[6] = 0) THEN TMP\_DEST[447:384] ← TMP\_SRC1[447:384]; FI;

IF (imm8[6] = 1) THEN TMP\_DEST[447:384] ← TMP\_SRC1[511:448]; FI;

IF (imm8[7] = 0) THEN TMP\_DEST[511:448] ← TMP\_SRC1[447:384]; FI;

IF (imm8[7] = 1) THEN TMP\_DEST[511:448] ← TMP\_SRC1[511:448]; FI;

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPERMILPD (256-bit immediate version)**

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]

IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]

IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]

IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]

IF (imm8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]

IF (imm8[2] = 1) THEN DEST[191:128] ← SRC1[255:192]

IF (imm8[3] = 0) THEN DEST[255:192] ← SRC1[191:128]

IF (imm8[3] = 1) THEN DEST[255:192] ← SRC1[255:192]

DEST[MAXVL-1:256] ← 0

**VPERMILPD (128-bit immediate version)**

```

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**VPERMILPD (EVEX variable versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN TMP\_SRC2[i+63:i] ← SRC2[63:0];

        ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i];

    FI;

ENDFOR;

IF (TMP\_SRC2[1] = 0) THEN TMP\_DEST[63:0] ← SRC1[63:0]; FI;

IF (TMP\_SRC2[1] = 1) THEN TMP\_DEST[63:0] ← SRC1[127:64]; FI;

IF (TMP\_SRC2[65] = 0) THEN TMP\_DEST[127:64] ← SRC1[63:0]; FI;

IF (TMP\_SRC2[65] = 1) THEN TMP\_DEST[127:64] ← SRC1[127:64]; FI;

IF VL ≥ 256

    IF (TMP\_SRC2[129] = 0) THEN TMP\_DEST[191:128] ← SRC1[191:128]; FI;

    IF (TMP\_SRC2[129] = 1) THEN TMP\_DEST[191:128] ← SRC1[255:192]; FI;

    IF (TMP\_SRC2[193] = 0) THEN TMP\_DEST[255:192] ← SRC1[191:128]; FI;

    IF (TMP\_SRC2[193] = 1) THEN TMP\_DEST[255:192] ← SRC1[255:192]; FI;

FI;

IF VL ≥ 512

    IF (TMP\_SRC2[257] = 0) THEN TMP\_DEST[319:256] ← SRC1[319:256]; FI;

    IF (TMP\_SRC2[257] = 1) THEN TMP\_DEST[319:256] ← SRC1[383:320]; FI;

    IF (TMP\_SRC2[321] = 0) THEN TMP\_DEST[383:320] ← SRC1[319:256]; FI;

    IF (TMP\_SRC2[321] = 1) THEN TMP\_DEST[383:320] ← SRC1[383:320]; FI;

    IF (TMP\_SRC2[385] = 0) THEN TMP\_DEST[447:384] ← SRC1[447:384]; FI;

    IF (TMP\_SRC2[385] = 1) THEN TMP\_DEST[447:384] ← SRC1[511:448]; FI;

    IF (TMP\_SRC2[449] = 0) THEN TMP\_DEST[511:448] ← SRC1[447:384]; FI;

    IF (TMP\_SRC2[449] = 1) THEN TMP\_DEST[511:448] ← SRC1[511:448]; FI;

FI;

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+63:i] remains unchanged\*

        ELSE ; zeroing-masking

            DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPERMILPD (256-bit variable version)**

```

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128] ← SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128] ← SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192] ← SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192] ← SRC1[255:192]
DEST[MAXVL-1:256] ← 0

```

**VPERMILPD (128-bit variable version)**

```

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMILPD __m512d __mm512_permute_pd( __m512d a, int imm);
VPERMILPD __m512d __mm512_mask_permute_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d __mm512_maskz_permute_pd( __mmask8 k, __m512d a, int imm);
VPERMILPD __m256d __mm256_mask_permute_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d __mm256_maskz_permute_pd( __mmask8 k, __m256d a, int imm);
VPERMILPD __m128d __mm_mask_permute_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VPERMILPD __m128d __mm_maskz_permute_pd( __mmask8 k, __m128d a, int imm);
VPERMILPD __m512d __mm512_permutevar_pd( __m512i i, __m512d a);
VPERMILPD __m512d __mm512_mask_permutevar_pd(__m512d s, __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m512d __mm512_maskz_permutevar_pd( __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m256d __mm256_mask_permutevar_pd(__m256d s, __mmask8 k, __m256i i, __m256d a);
VPERMILPD __m256d __mm256_maskz_permutevar_pd( __mmask8 k, __m256i i, __m256d a);
VPERMILPD __m128d __mm_mask_permutevar_pd(__m128d s, __mmask8 k, __m128i i, __m128d a);
VPERMILPD __m128d __mm_maskz_permutevar_pd( __mmask8 k, __m128i i, __m128d a);
VPERMILPD __m128d __mm_permute_pd( __m128d a, int control)
VPERMILPD __m256d __mm256_permute_pd( __m256d a, int control)
VPERMILPD __m128d __mm_permutevar_pd( __m128d a, __m128i control);
VPERMILPD __m256d __mm256_permutevar_pd( __m256d a, __m256i control);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If either (E)VEX.vvvv != 1111B and with imm8.

## VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute single-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute single-precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.
VEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute single-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
VEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute single-precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.
EVEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 0C /r VPERMILPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.
EVEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	D	V/V	AVX512VL AVX512F	Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	D	V/V	AVX512VL AVX512F	Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W0 04 /r ibVPERMILPS zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	D	V/V	AVX512F	Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

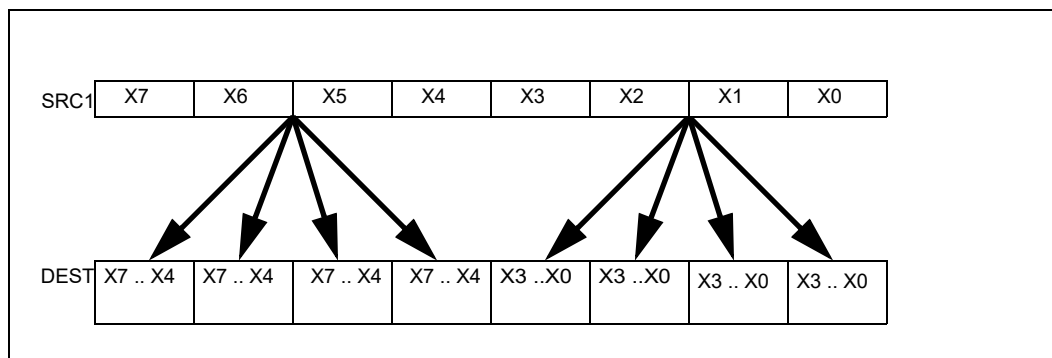
**Description**

(variable control version)

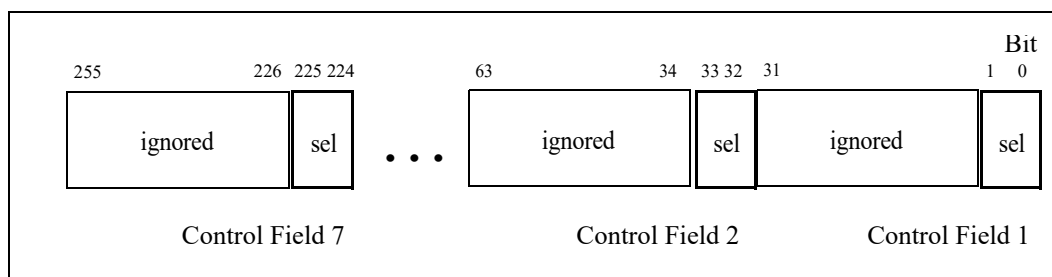
Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 5-26). Each control determines which of the source element in an input quadruple is selected for the destination element. Each quadruple of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.



**Figure 5-25. VPERMILPS Operation**



**Figure 5-26. VPERMILPS Shuffle Control**

(immediate control version)

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

Note: For the imm8 version, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

**Operation**

```

Select4(SRC, control) {
CASE (control[1:0]) OF
  0:  TMP ← SRC[31:0];
  1:  TMP ← SRC[63:32];
  2:  TMP ← SRC[95:64];
  3:  TMP ← SRC[127:96];
ESAC;
RETURN TMP
}

```

**VPERMILPS (EVEX immediate versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

    THEN TMP\_SRC1[i+31:i] ← SRC1[31:0];

    ELSE TMP\_SRC1[i+31:i] ← SRC1[i+31:i];

  FI;

ENDFOR;

TMP\_DEST[31:0] ← Select4(TMP\_SRC1[127:0], imm8[1:0]);

TMP\_DEST[63:32] ← Select4(TMP\_SRC1[127:0], imm8[3:2]);

TMP\_DEST[95:64] ← Select4(TMP\_SRC1[127:0], imm8[5:4]);

TMP\_DEST[127:96] ← Select4(TMP\_SRC1[127:0], imm8[7:6]); FI;

IF VL ≥ 256

  TMP\_DEST[159:128] ← Select4(TMP\_SRC1[255:128], imm8[1:0]); FI;

  TMP\_DEST[191:160] ← Select4(TMP\_SRC1[255:128], imm8[3:2]); FI;

  TMP\_DEST[223:192] ← Select4(TMP\_SRC1[255:128], imm8[5:4]); FI;

  TMP\_DEST[255:224] ← Select4(TMP\_SRC1[255:128], imm8[7:6]); FI;

FI;

IF VL ≥ 512

  TMP\_DEST[287:256] ← Select4(TMP\_SRC1[383:256], imm8[1:0]); FI;

  TMP\_DEST[319:288] ← Select4(TMP\_SRC1[383:256], imm8[3:2]); FI;

  TMP\_DEST[351:320] ← Select4(TMP\_SRC1[383:256], imm8[5:4]); FI;

  TMP\_DEST[383:352] ← Select4(TMP\_SRC1[383:256], imm8[7:6]); FI;

  TMP\_DEST[415:384] ← Select4(TMP\_SRC1[511:384], imm8[1:0]); FI;

  TMP\_DEST[447:416] ← Select4(TMP\_SRC1[511:384], imm8[3:2]); FI;

  TMP\_DEST[479:448] ← Select4(TMP\_SRC1[511:384], imm8[5:4]); FI;

  TMP\_DEST[511:480] ← Select4(TMP\_SRC1[511:384], imm8[7:6]); FI;

FI;

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ← TMP\_DEST[i+31:i]

  ELSE

    IF \*merging-masking\*

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE DEST[i+31:i] ← 0 ;zeroing-masking

  FI;

  FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPERMILPS (256-bit immediate version)**

```

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ← Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC1[255:128], imm8[7:6]);

```

**VPERMILPS (128-bit immediate version)**

```

DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[MAXVL-1:128] ← 0

```

**VPERMILPS (EVEX variable versions)**

```

(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0];
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i];
    FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], TMP_SRC2[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], TMP_SRC2[33:32]);
TMP_DEST[95:64] ← Select4(SRC1[127:0], TMP_SRC2[65:64]);
TMP_DEST[127:96] ← Select4(SRC1[127:0], TMP_SRC2[97:96]);
IF VL >= 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], TMP_SRC2[129:128]);
    TMP_DEST[191:160] ← Select4(SRC1[255:128], TMP_SRC2[161:160]);
    TMP_DEST[223:192] ← Select4(SRC1[255:128], TMP_SRC2[193:192]);
    TMP_DEST[255:224] ← Select4(SRC1[255:128], TMP_SRC2[225:224]);
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], TMP_SRC2[257:256]);
    TMP_DEST[319:288] ← Select4(SRC1[383:256], TMP_SRC2[289:288]);
    TMP_DEST[351:320] ← Select4(SRC1[383:256], TMP_SRC2[321:320]);
    TMP_DEST[383:352] ← Select4(SRC1[383:256], TMP_SRC2[353:352]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], TMP_SRC2[385:384]);
    TMP_DEST[447:416] ← Select4(SRC1[511:384], TMP_SRC2[417:416]);
    TMP_DEST[479:448] ← Select4(SRC1[511:384], TMP_SRC2[449:448]);
    TMP_DEST[511:480] ← Select4(SRC1[511:384], TMP_SRC2[481:480]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
            ELSE DEST[i+31:i] ← 0 ;zeroing-masking
        FI
    FI
ENDFOR

```

```

        FI;
    ENDFOR
    DEST[MAXVL-1:VL] ← 0

```

**VPERMILPS (256-bit variable version)**

```

DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] ← Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] ← Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] ← Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] ← Select4(SRC1[255:128], SRC2[225:224]);
DEST[MAXVL-1:256] ← 0

```

**VPERMILPS (128-bit variable version)**

```

DEST[31:0] ← Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ← Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ← Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMILPS __m512 __mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 __mm512_mask_permute_ps( __m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 __mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 __mm256_mask_permute_ps( __m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 __mm_mask_permute_ps( __m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 __mm_maskz_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m512 __mm512_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 __mm512_mask_permutevar_ps( __m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m512 __mm512_maskz_permutevar_ps( __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m256 __mm256_mask_permutevar_ps( __m256 s, __mmask8 k, __m256i i, __m256 a);
VPERMILPS __m256 __mm256_maskz_permutevar_ps( __mmask8 k, __m256i i, __m256 a);
VPERMILPS __m128 __mm_mask_permutevar_ps( __m128 s, __mmask8 k, __m128i i, __m128 a);
VPERMILPS __m128 __mm_maskz_permutevar_ps( __mmask8 k, __m128i i, __m128 a);
VPERMILPS __m128 __mm_permute_ps( __m128 a, int control);
VPERMILPS __m256 __mm256_permute_ps( __m256 a, int control);
VPERMILPS __m128 __mm_permutevar_ps( __m128 a, __m128i control);
VPERMILPS __m256 __mm256_permutevar_ps( __m256 a, __m256i control);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4;

#UD If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If either (E)VEX.vvvv != 1111B and with imm8.



## VPERMPD—Permute Double-Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1, ymm2/m256, imm8	A	V/V	AVX2	Permute double-precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	AVX512VL AVX512F	Permute double-precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W1 01 /r ib VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	B	V/V	AVX512F	Permute double-precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 16 /r VPERMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Permute double-precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 16 /r VPERMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Permute double-precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

The imm8 version: Copies quadword elements of double-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

**Operation****VPERMPD (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC \*is memory\*)  
 THEN TMP\_SRC[i+63:i] ← SRC[63:0];  
 ELSE TMP\_SRC[i+63:i] ← SRC[i+63:i];

FI;

ENDFOR;

TMP\_DEST[63:0] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[1:0] \* 64))[63:0];

TMP\_DEST[127:64] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[3:2] \* 64))[63:0];

TMP\_DEST[191:128] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[5:4] \* 64))[63:0];

TMP\_DEST[255:192] ← (TMP\_SRC[256:0] &gt;&gt; (IMM8[7:6] \* 64))[63:0];

IF VL &gt;= 512

TMP\_DEST[319:256] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[1:0] \* 64))[63:0];

TMP\_DEST[383:320] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[3:2] \* 64))[63:0];

TMP\_DEST[447:384] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[5:4] \* 64))[63:0];

TMP\_DEST[511:448] ← (TMP\_SRC[511:256] &gt;&gt; (IMM8[7:6] \* 64))[63:0];

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ;merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ;zeroing-masking

DEST[i+63:i] ← 0 ;zeroing-masking

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPERMPD (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)  
 THEN TMP\_SRC2[i+63:i] ← SRC2[63:0];  
 ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i];

FI;

ENDFOR;

IF VL = 256

TMP\_DEST[63:0] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[1:0] \* 64))[63:0];

TMP\_DEST[127:64] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[65:64] \* 64))[63:0];

TMP\_DEST[191:128] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[129:128] \* 64))[63:0];

TMP\_DEST[255:192] ← (TMP\_SRC2[255:0] &gt;&gt; (SRC1[193:192] \* 64))[63:0];

FI;

IF VL = 512

TMP\_DEST[63:0] ← (TMP\_SRC2[511:0] &gt;&gt; (SRC1[2:0] \* 64))[63:0];

```

    TMP_DEST[127:64] ← (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[191:128] ← (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
    TMP_DEST[255:192] ← (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
    TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
    TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0 ;zeroing-masking
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

#### VPERMPD (VEX.256 encoded version)

```

DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VPERMPD __m512d __mm512_permutex_pd( __m512d a, int imm);
VPERMPD __m512d __mm512_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d __mm512_maskz_permutex_pd( __mmask16 k, __m512d a, int imm);
VPERMPD __m512d __mm512_permutexvar_pd( __m512i i, __m512d a);
VPERMPD __m512d __mm512_mask_permutexvar_pd(__m512d s, __mmask16 k, __m512i i, __m512d a);
VPERMPD __m512d __mm512_maskz_permutexvar_pd( __mmask16 k, __m512i i, __m512d a);
VPERMPD __m256d __mm256_permutex_epi64( __m256d a, int imm);
VPERMPD __m256d __mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256d a, int imm);
VPERMPD __m256d __mm256_maskz_permutex_epi64( __mmask8 k, __m256d a, int imm);
VPERMPD __m256d __mm256_permutexvar_epi64( __m256i i, __m256d a);
VPERMPD __m256d __mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d __mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i i, __m256d a);

```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD If VEX.L = 0.  
If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If encoded with EVEX.128.  
If EVEX.vvvv != 1111B and with imm8.

## VPERMPS—Permute Single-Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 16 /r VPERMPS ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Permute single-precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.NDS.256.66.0F38.W0 16 /r VPERMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision floating-point elements in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 subject to write mask k1.
EVEX.NDS.512.66.0F38.W0 16 /r VPERMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Copies doubleword elements of single-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### VPERMPS (EVEX forms)

(KL, VL) (8, 256),= (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

    THEN TMP\_SRC2[i+31:i] ← SRC2[31:0];

    ELSE TMP\_SRC2[i+31:i] ← SRC2[i+31:i];

  FI;

ENDFOR;

IF VL = 256

  TMP\_DEST[31:0] ← (TMP\_SRC2[255:0] >> (SRC1[2:0] \* 32))[31:0];

  TMP\_DEST[63:32] ← (TMP\_SRC2[255:0] >> (SRC1[34:32] \* 32))[31:0];

  TMP\_DEST[95:64] ← (TMP\_SRC2[255:0] >> (SRC1[66:64] \* 32))[31:0];

  TMP\_DEST[127:96] ← (TMP\_SRC2[255:0] >> (SRC1[98:96] \* 32))[31:0];

  TMP\_DEST[159:128] ← (TMP\_SRC2[255:0] >> (SRC1[130:128] \* 32))[31:0];

  TMP\_DEST[191:160] ← (TMP\_SRC2[255:0] >> (SRC1[162:160] \* 32))[31:0];

  TMP\_DEST[223:192] ← (TMP\_SRC2[255:0] >> (SRC1[193:192] \* 32))[31:0];

  TMP\_DEST[255:224] ← (TMP\_SRC2[255:0] >> (SRC1[226:224] \* 32))[31:0];

```

FI;
IF VL = 512
    TMP_DEST[31:0] ← (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
    TMP_DEST[63:32] ← (TMP_SRC2[511:0] >> (SRC1[35:32] * 32))[31:0];
    TMP_DEST[95:64] ← (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
    TMP_DEST[127:96] ← (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
    TMP_DEST[159:128] ← (TMP_SRC2[511:0] >> (SRC1[131:128] * 32))[31:0];
    TMP_DEST[191:160] ← (TMP_SRC2[511:0] >> (SRC1[163:160] * 32))[31:0];
    TMP_DEST[223:192] ← (TMP_SRC2[511:0] >> (SRC1[195:192] * 32))[31:0];
    TMP_DEST[255:224] ← (TMP_SRC2[511:0] >> (SRC1[227:224] * 32))[31:0];
    TMP_DEST[287:256] ← (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
    TMP_DEST[319:288] ← (TMP_SRC2[511:0] >> (SRC1[291:288] * 32))[31:0];
    TMP_DEST[351:320] ← (TMP_SRC2[511:0] >> (SRC1[323:320] * 32))[31:0];
    TMP_DEST[383:352] ← (TMP_SRC2[511:0] >> (SRC1[355:352] * 32))[31:0];
    TMP_DEST[415:384] ← (TMP_SRC2[511:0] >> (SRC1[387:384] * 32))[31:0];
    TMP_DEST[447:416] ← (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
    TMP_DEST[479:448] ← (TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
    TMP_DEST[511:480] ← (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ;merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ;zeroing-masking
            DEST[i+31:i] ← 0 ;zeroing-masking
    FI;
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPERMPS (VEX.256 encoded version)**

```

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMPS __m512 __mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 __mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 __mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 __mm256_permutexvar_ps(__m256i i, __m256 a);
VPERMPS __m256 __mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256i i, __m256 a);
VPERMPS __m256 __mm256_maskz_permutexvar_ps(__mmask8 k, __m256i i, __m256 a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E4NF.

## VPERMQ—Qwords Element Permutation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1, ymm2/m256, imm8	A	V/V	AVX2	Permute qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	AVX512VL AVX512F	Permute qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1.
EVEX.512.66.0F3A.W1 00 /r ib VPERMQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	B	V/V	AVX512F	Permute qwords in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1.
EVEX.NDS.256.66.0F38.W1 36 /r VPERMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1.
EVEX.NDS.512.66.0F38.W1 36 /r VPERMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.

**Operation****VPERMQ (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC \*is memory\*)  
 THEN TMP\_SRC[i+63:i] ← SRC[63:0];  
 ELSE TMP\_SRC[i+63:i] ← SRC[i+63:i];

FI;

ENDFOR;

TMP\_DEST[63:0] ← (TMP\_SRC[255:0] >> (IMM8[1:0] \* 64))[63:0];  
 TMP\_DEST[127:64] ← (TMP\_SRC[255:0] >> (IMM8[3:2] \* 64))[63:0];  
 TMP\_DEST[191:128] ← (TMP\_SRC[255:0] >> (IMM8[5:4] \* 64))[63:0];  
 TMP\_DEST[255:192] ← (TMP\_SRC[255:0] >> (IMM8[7:6] \* 64))[63:0];

IF VL &gt;= 512

TMP\_DEST[319:256] ← (TMP\_SRC[511:256] >> (IMM8[1:0] \* 64))[63:0];  
 TMP\_DEST[383:320] ← (TMP\_SRC[511:256] >> (IMM8[3:2] \* 64))[63:0];  
 TMP\_DEST[447:384] ← (TMP\_SRC[511:256] >> (IMM8[5:4] \* 64))[63:0];  
 TMP\_DEST[511:448] ← (TMP\_SRC[511:256] >> (IMM8[7:6] \* 64))[63:0];

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*  
 THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]  
 ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0 ;zeroing-masking

FI;

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPERMQ (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)  
 THEN TMP\_SRC2[i+63:i] ← SRC2[63:0];  
 ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i];

FI;

ENDFOR;

IF VL = 256

TMP\_DEST[63:0] ← (TMP\_SRC2[255:0] >> (SRC1[1:0] \* 64))[63:0];  
 TMP\_DEST[127:64] ← (TMP\_SRC2[255:0] >> (SRC1[65:64] \* 64))[63:0];  
 TMP\_DEST[191:128] ← (TMP\_SRC2[255:0] >> (SRC1[129:128] \* 64))[63:0];  
 TMP\_DEST[255:192] ← (TMP\_SRC2[255:0] >> (SRC1[193:192] \* 64))[63:0];

FI;

IF VL = 512

TMP\_DEST[63:0] ← (TMP\_SRC2[511:0] >> (SRC1[2:0] \* 64))[63:0];  
 TMP\_DEST[127:64] ← (TMP\_SRC2[511:0] >> (SRC1[66:64] \* 64))[63:0];  
 TMP\_DEST[191:128] ← (TMP\_SRC2[511:0] >> (SRC1[130:128] \* 64))[63:0];  
 TMP\_DEST[255:192] ← (TMP\_SRC2[511:0] >> (SRC1[194:192] \* 64))[63:0];



```

    TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
    TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0 ;zeroing-masking
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

#### VPERMQ (VEX.256 encoded version)

```

DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ← (SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VPERMQ __m512i _mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i _mm512_mask_permutex_epi64( __m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_maskz_permutex_epi64( __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i _mm512_mask_permutexvar_epi64( __m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i _mm512_maskz_permutexvar_epi64( __mmask8 k, __m512i a, __m512i b);
VPERMQ __m256i _mm256_permutex_epi64( __m256i a, int imm);
VPERMQ __m256i _mm256_mask_permutex_epi64( __m256i s, __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_maskz_permutex_epi64( __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_permutexvar_epi64( __m256i a, __m256i b);
VPERMQ __m256i _mm256_mask_permutexvar_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b);
VPERMQ __m256i _mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i a, __m256i b);

```

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD If VEX.L = 0.  
If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If encoded with EVEX.128.  
If EVEX.vvvv != 1111B and with imm8.

## VPERMT2B—Full Permute of Bytes from Two Tables Overwriting a Table

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W0 7D /r VPERMT2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512_VBMI	Permute bytes in xmm3/m128 and xmm1 using byte indexes in xmm2 and store the byte results in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 7D /r VPERMT2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512_VBMI	Permute bytes in ymm3/m256 and ymm1 using byte indexes in ymm2 and store the byte results in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 7D /r VPERMT2B zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI	Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Permutes byte values from two tables, comprising of the first operand (also the destination operand) and the third operand (the second source operand). The second operand (the first source operand) provides byte indices to select byte results from the two tables. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result. The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the second table and the indices can be reused in subsequent iterations, but the first table is overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****VPERMT2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

id ← 3;

ELSE IF VL = 256:

id ← 4;

ELSE IF VL = 512:

id ← 5;

FI;

TMP\_DEST[VL-1:0] ← DEST[VL-1:0];

FOR j ← 0 TO KL-1

off ← 8\*SRC1[j\*8 + id:j\*8];

IF k1[j] OR \*no writemask\*:

DEST[j\*8 + 7:j\*8] ← SRC1[j\*8+id+1]? SRC2[off+7:off] : TMP\_DEST[off+7:off];

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7:j\*8] ← 0;

\*ELSE

DEST[j\*8 + 7:j\*8] remains unchanged\*

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMT2B \_\_m512i \_\_mm512\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m512i \_\_mm512\_mask\_permutex2var\_epi8(\_\_m512i a, \_\_mmask64 k, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m256i \_\_mm256\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m256i \_\_mm256\_mask\_permutex2var\_epi8(\_\_m256i a, \_\_mmask32 k, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m128i \_\_mm\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMT2B \_\_m128i \_\_mm\_mask\_permutex2var\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m128i idx, \_\_m128i b);

VPERMT2B \_\_m128i \_\_mm\_maskz\_permutex2var\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.

**VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 7D /r VPERMT2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 7D /r VPERMT2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7D /r VPERMT2W zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW	Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 7E /r VPERMT2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 7E /r VPERMT2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 7E /r VPERMT2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W1 7E /r VPERMT2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 7E /r VPERMT2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7E /r VPERMT2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 7F /r VPERMT2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 7F /r VPERMT2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 7F /r VPERMT2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W1 7F /r VPERMT2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 7F /r VPERMT2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7F /r VPERMT2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Permutates 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table\_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table\_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

## Operation

**VPERMT2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

id ← 2

FI;

IF VL = 256

id ← 3

FI;

IF VL = 512

id ← 4

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 16

off ← 16\*SRC1[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

DEST[i+15:i]=SRC1[i+id+1] ? SRC2[off+15:off]

: TMP\_DEST[off+15:off]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

```

        DEST[i+15:i] ← 0
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPERMT2D/VPERMT2PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

IF VL = 128

id ← 1

FI;

IF VL = 256

id ← 2

FI;

IF VL = 512

id ← 3

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 32

off ← 32 \* SRC1[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[31:0]

: TMP\_DEST[off+31:off]

ELSE

DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[off+31:off]

: TMP\_DEST[off+31:off]

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPERMT2Q/VPERMT2PD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL = 128

id ← 0

FI;

IF VL = 256

id ← 1

FI;

IF VL = 512

id ← 2

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

```

i ← j * 64
off ← 64 * SRC1[i+id:i]
IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[63:0]
                : TMP_DEST[off+63:off]
            ELSE
                DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[off+63:off]
                : TMP_DEST[off+63:off]
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VPERMT2D __m512i __mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMT2D __m512i __mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMT2D __m512i __mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMT2D __m512i __mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMT2D __m256i __mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMT2D __m256i __mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2D __m256i __mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2D __m256i __mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2D __m128i __mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMT2D __m128i __mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2D __m128i __mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2D __m128i __mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2PD __m512d __mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMT2PD __m512d __mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMT2PD __m512d __mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMT2PD __m512d __mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMT2PD __m256d __mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMT2PD __m256d __mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMT2PD __m256d __mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMT2PD __m256d __mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMT2PD __m128d __mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMT2PD __m128d __mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMT2PD __m128d __mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMT2PD __m128d __mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMT2PS __m512 __mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMT2PS __m512 __mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMT2PS __m512 __mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMT2PS __m512 __mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);

```

VPERMT2PS \_\_m256 \_\_mm256\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_mask\_permutex2var\_ps(\_\_m256 a, \_\_mmask8 k, \_\_m256i idx, \_\_m256 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_mask2\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_mmask8 k, \_\_m256 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMT2PS \_\_m128 \_\_mm\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMT2PS \_\_m128 \_\_mm\_mask\_permutex2var\_ps(\_\_m128 a, \_\_mmask8 k, \_\_m128i idx, \_\_m128 b);  
 VPERMT2PS \_\_m128 \_\_mm\_mask2\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_mmask8 k, \_\_m128 b);  
 VPERMT2PS \_\_m128 \_\_mm\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMT2Q \_\_m512i \_\_mm512\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2Q \_\_m512i \_\_mm512\_mask\_permutex2var\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i idx, \_\_m512i b);  
 VPERMT2Q \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_mmask8 k, \_\_m512i b);  
 VPERMT2Q \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_mask\_permutex2var\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2Q \_\_m128i \_\_mm\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2Q \_\_m128i \_\_mm\_mask\_permutex2var\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMT2Q \_\_m128i \_\_mm\_mask2\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMT2Q \_\_m128i \_\_mm\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2W \_\_m512i \_\_mm512\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2W \_\_m512i \_\_mm512\_mask\_permutex2var\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m512i idx, \_\_m512i b);  
 VPERMT2W \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_mmask32 k, \_\_m512i b);  
 VPERMT2W \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2W \_\_m256i \_\_mm256\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2W \_\_m256i \_\_mm256\_mask\_permutex2var\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m256i idx, \_\_m256i b);  
 VPERMT2W \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_mmask16 k, \_\_m256i b);  
 VPERMT2W \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2W \_\_m128i \_\_mm\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2W \_\_m128i \_\_mm\_mask\_permutex2var\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMT2W \_\_m128i \_\_mm\_mask2\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMT2W \_\_m128i \_\_mm\_maskz\_permutex2var\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

VPERMT2D/Q/PS/PD: See Exceptions Type E4NF.

VPERMT2W: See Exceptions Type E4NF.nb.



## VPEXPANDD—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 89 /r VPEXPANDD xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 89 /r VPEXPANDD ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 89 /r VPEXPANDD zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F	Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VPEXPANDD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

k ← 0

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN

            DEST[i+31:i] ← SRC[k+31:k];

            k ← k + 32

        ELSE

            IF \*merging-masking\* ; merging-masking

                THEN \*DEST[i+31:i] remains unchanged\*

                ELSE ; zeroing-masking

                    DEST[i+31:i] ← 0

        FI

    FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPEXPANDD __m512i _mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_maskz_expandloadu_epi32(__mmask16 k, void * a);
VPEXPANDD __m512i _mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VPEXPANDD __m512i _mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
VPEXPANDD __m256i _mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m256i _mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i _mm256_maskz_expand_epi32(__mmask8 k, __m256i a);
VPEXPANDD __m128i _mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i _mm_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m128i _mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VPEXPANDD __m128i _mm_maskz_expand_epi32(__mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.

## VPEXPANDQ—Load Sparse Packed Quadword Integer Values from Dense Memory / Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 89 /r VPEXPANDQ xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 89 /r VPEXPANDQ ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 89 /r VPEXPANDQ zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F	Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VPEXPANDQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

$k \leftarrow 0$

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

IF  $k1[j]$  OR \*no writemask\*

THEN

$DEST[i+63:i] \leftarrow SRC[k+63:k];$

$k \leftarrow k + 64$

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

THEN  $DEST[i+63:i] \leftarrow 0$

FI

FI;

ENDFOR

$DEST[MAXVL-1:VL] \leftarrow 0$

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPEXPANDQ __m512i _mm512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_mask_expand_epi64(__m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m512i _mm512_maskz_expand_epi64(__mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_mask_expand_epi64(__m256i s, __mmask8 k, __m256i a);
VPEXPANDQ __m256i _mm256_maskz_expand_epi64(__mmask8 k, __m256i a);
VPEXPANDQ __m128i _mm_mask_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m128i _mm_mask_expand_epi64(__m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm_maskz_expand_epi64(__mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD                      If EVEX.vvvv != 1111B.

## VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 90 /r VPGATHERDD <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W0 91 /r VPGATHERQD <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W0 90 /r VPGATHERDD <i>ymm1</i> , <i>vm32y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32y</i> , gather dword from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W0 91 /r VPGATHERQD <i>xmm1</i> , <i>vm64y</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

### Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST  $\leftarrow$  SRC1;

BASE\_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK  $\leftarrow$  SRC3;

### VPGATHERDD (VEX.128 version)

FOR j  $\leftarrow$  0 to 3

    i  $\leftarrow$  j \* 32;

    IF MASK[31:i] THEN

        MASK[i + 31:i]  $\leftarrow$  FFFFFFFFH; // extend from most significant bit

    ELSE

        MASK[i + 31:i]  $\leftarrow$  0;

    FI;

ENDFOR

MASK[MAXVL-1:128]  $\leftarrow$  0;

FOR j  $\leftarrow$  0 to 3

    i  $\leftarrow$  j \* 32;

    DATA\_ADDR  $\leftarrow$  BASE\_ADDR + (SignExtend(VINDEX[i+31:i])\*SCALE + DISP;

    IF MASK[31:i] THEN

        DEST[i + 31:i]  $\leftarrow$  FETCH\_32BITS(DATA\_ADDR); // a fault exits the instruction

    FI;

    MASK[i + 31:i]  $\leftarrow$  0;

ENDFOR

DEST[MAXVL-1:128]  $\leftarrow$  0;

(non-masked elements of the mask register have the content of respective element cleared)

**VPGATHERQD (VEX.128 version)**

```

FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 1
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[MAXVL-1:64] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

**VPGATHERDD (VEX.256 version)**

```

FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 7
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

**VPGATHERQD (VEX.256 version)**

```

FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31:i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31:i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPGATHERDD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERDD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERDD: __m256i _mm256_i32gather_epi32 ( int const * base, __m256i index, const int scale);
VPGATHERDD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);
VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERQD: __m128i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);
VPGATHERQD: __m128i _mm256_mask_i64gather_epi32 (__m128i src, int const * base, __m256i index, __m128i mask, const int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 12.



## VPATHERDD/VPATHERDQ—Gather Packed Dword, Packed Qword with Signed Dword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 90 /vsib VPATHERDD xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 90 /vsib VPATHERDD ymm1 {k1}, vm32y	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 90 /vsib VPATHERDD zmm1 {k1}, vm32z	A	V/V	AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 90 /vsib VPATHERDQ xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 90 /vsib VPATHERDQ ymm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 90 /vsib VPATHERDQ zmm1 {k1}, vm32y	A	V/V	AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

### Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address `BASE_ADDR` and index vector `VINDEX` with scale `SCALE` are gathered. The result is written into vector `zmm1`. The elements are specified via the VSIB (i.e., the index register is a `zmm`, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register (`zmm1`) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (`k1`) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination `zmm` will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a `#UD` fault.
- These instructions do not accept zeroing-masking since the 0 values in `k1` are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same  $\text{disp8} \times N$  and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VPGATHERDD (EVEX encoded version)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j]

    THEN DEST[i+31:i] ← MEM[BASE\_ADDR +  
      SignExtend(VINDEX[i+31:i]) \* SCALE + DISP], 1)

    k1[j] ← 0

    ELSE \*DEST[i+31:i] ← remains unchanged\*                   ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAXVL-1:VL] ← 0

#### VPGATHERDQ (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j]

    THEN DEST[i+63:i] ←  
      MEM[BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP]

    k1[j] ← 0

    ELSE \*DEST[i+63:i] ← remains unchanged\*                   ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPGATHERDD __m512i _mm512_i32gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERDD __m512i _mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i _mm256_mask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i _mm_mask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_i32logather_epi64(__m256i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDQ __m256i _mm256_mask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i _mm_mask_i32gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

## VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 90 /r VPGATHERDQ <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	A	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather qword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.128.66.0F38.W1 91 /r VPGATHERQQ <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	A	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather qword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.DDS.256.66.0F38.W1 90 /r VPGATHERDQ <i>ymm1</i> , <i>vm32x</i> , <i>ymm2</i>	A	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather qword values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.DDS.256.66.0F38.W1 91 /r VPGATHERQQ <i>ymm1</i> , <i>vm64y</i> , <i>ymm2</i>	A	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather qword values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

### Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST  $\leftarrow$  SRC1;

BASE\_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK  $\leftarrow$  SRC3;

### VPGATHERDQ (VEX.128 version)

FOR j  $\leftarrow$  0 to 1

    i  $\leftarrow$  j \* 64;

    IF MASK[63:i] THEN

        MASK[i + 63:i]  $\leftarrow$  FFFFFFFF\_FFFFFFFFH; // extend from most significant bit

    ELSE

        MASK[i + 63:i]  $\leftarrow$  0;

    FI;

ENDFOR

FOR j  $\leftarrow$  0 to 1

    k  $\leftarrow$  j \* 32;

    i  $\leftarrow$  j \* 64;

    DATA\_ADDR  $\leftarrow$  BASE\_ADDR + (SignExtend(VINDEX[k+31:k])\*SCALE + DISP;

    IF MASK[63:i] THEN

        DEST[i + 63:i]  $\leftarrow$  FETCH\_64BITS(DATA\_ADDR); // a fault exits the instruction

    FI;

    MASK[i + 63:i]  $\leftarrow$  0;

ENDFOR

MASK[MAXVL-1:128]  $\leftarrow$  0;

DEST[MAXVL-1:128]  $\leftarrow$  0;

(non-masked elements of the mask register have the content of respective element cleared)

#### VPGATHERQQ (VEX.128 version)

```

FOR j ← 0 to 1
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;

```

(non-masked elements of the mask register have the content of respective element cleared)

#### VPGATHERQQ (VEX.256 version)

```

FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR

```

(non-masked elements of the mask register have the content of respective element cleared)

#### VPGATHERDQ (VEX.256 version)

```

FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  k ← j * 32;
  i ← j * 64;

```

```

DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
IF MASK[63:i] THEN
    DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[i +63:i] ← 0;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPGATHERDQ: __m128i _mm_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);
VPGATHERDQ: __m128i _mm_mask_i32gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int
scale);
VPGATHERDQ: __m256i _mm256_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);
VPGATHERDQ: __m256i _mm256_mask_i32gather_epi64 (__m256i src, __int64 const * base, __m128i index, __m256i mask, const
int scale);
VPGATHERQQ: __m128i _mm_i64gather_epi64 (__int64 const * base, __m128i index, const int scale);
VPGATHERQQ: __m128i _mm_mask_i64gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int
scale);
VPGATHERQQ: __m256i _mm256_i64gather_epi64 (__int64 const * base, __m256i index, const int scale);
VPGATHERQQ: __m256i _mm256_mask_i64gather_epi64 (__m256i src, __int64 const * base, __m256i index, __m256i mask, const
int scale);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 12.

## VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 91 /vsib VPGATHERQD ymm1 {k1}, vm64z	A	V/V	AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 91 /vsib VPGATHERQQ xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 91 /vsib VPGATHERQQ ymm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 91 /vsib VPGATHERQQ zmm1 {k1}, vm64z	A	V/V	AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

### Description

A set of 8 doubleword/quadword memory locations pointed to by base address `BASE_ADDR` and index vector `VINDEX` with scale `SCALE` are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.



- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same  $\text{disp8} \cdot N$  and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VPGATHERQD (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j  $\leftarrow$  0 TO KL-1

    i  $\leftarrow$  j \* 32

    k  $\leftarrow$  j \* 64

    IF k1[j]

        THEN DEST[i+31:i]  $\leftarrow$  MEM[BASE\_ADDR + (VINDEX[k+63:k] \* SCALE + DISP)], 1)

        k1[j]  $\leftarrow$  0

    ELSE \*DEST[i+31:i]  $\leftarrow$  remains unchanged\* ; Only merging masking is allowed

    FI;

ENDFOR

k1[MAX\_KL-1:KL]  $\leftarrow$  0

DEST[MAXVL-1:VL/2]  $\leftarrow$  0

#### VPGATHERQQ (EVEX encoded version)

(KL, VL) = (2, 64), (4, 128), (8, 256)

FOR j  $\leftarrow$  0 TO KL-1

    i  $\leftarrow$  j \* 64

    IF k1[j]

        THEN DEST[i+63:i]  $\leftarrow$

        MEM[BASE\_ADDR + (VINDEX[i+63:i] \* SCALE + DISP)]

        k1[j]  $\leftarrow$  0

    ELSE \*DEST[i+63:i]  $\leftarrow$  remains unchanged\* ; Only merging masking is allowed

    FI;

ENDFOR

k1[MAX\_KL-1:KL]  $\leftarrow$  0

DEST[MAXVL-1:VL]  $\leftarrow$  0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPGATHERQD __m256i _mm512_i64gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERQD __m256i _mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i _mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i _mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_i64gather_epi64(__m512i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQQ __m256i _mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQQ __m128i _mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

**VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 44 /r VPLZCNTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 44 /r VPLZCNTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 44 /r VPLZCNTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD	Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 44 /r VPLZCNTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 44 /r VPLZCNTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 44 /r VPLZCNTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD	Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

**Description**

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPLZCNTD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j\*32

IF MaskBit(j) OR \*no writemask\*

THEN

temp ← 32

DEST[i+31:i] ← 0

WHILE (temp &gt; 0) AND (SRC[i+temp-1] = 0)

DO

temp ← temp - 1

DEST[i+31:i] ← DEST[i+31:i] + 1

OD

ELSE

IF \*merging-masking\*

THEN \*DEST[i+31:i] remains unchanged\*

ELSE DEST[i+31:i] ← 0

FI

FI

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPLZCNTQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j\*64

IF MaskBit(j) OR \*no writemask\*

THEN

temp ← 64

DEST[i+63:i] ← 0

WHILE (temp &gt; 0) AND (SRC[i+temp-1] = 0)

DO

temp ← temp - 1

DEST[i+63:i] ← DEST[i+63:i] + 1

OD

ELSE

IF \*merging-masking\*

THEN \*DEST[i+63:i] remains unchanged\*

ELSE DEST[i+63:i] ← 0

FI

FI

ENDFOR

DEST[MAXVL-1:VL] ← 0

### Intel C/C++ Compiler Intrinsic Equivalent

```

VPLZCNTD __m512i _mm512_lzcnt_epi32(__m512i a);
VPLZCNTD __m512i _mm512_mask_lzcnt_epi32(__m512i s, __mmask16 m, __m512i a);
VPLZCNTD __m512i _mm512_maskz_lzcnt_epi32(__mmask16 m, __m512i a);
VPLZCNTQ __m512i _mm512_lzcnt_epi64(__m512i a);
VPLZCNTQ __m512i _mm512_mask_lzcnt_epi64(__m512i s, __mmask8 m, __m512i a);
VPLZCNTQ __m512i _mm512_maskz_lzcnt_epi64(__mmask8 m, __m512i a);
VPLZCNTD __m256i _mm256_lzcnt_epi32(__m256i a);
VPLZCNTD __m256i _mm256_mask_lzcnt_epi32(__m256i s, __mmask8 m, __m256i a);
VPLZCNTD __m256i _mm256_maskz_lzcnt_epi32(__mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_lzcnt_epi64(__m256i a);
VPLZCNTQ __m256i _mm256_mask_lzcnt_epi64(__m256i s, __mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_maskz_lzcnt_epi64(__mmask8 m, __m256i a);
VPLZCNTD __m128i _mm_lzcnt_epi32(__m128i a);
VPLZCNTD __m128i _mm_mask_lzcnt_epi32(__m128i s, __mmask8 m, __m128i a);
VPLZCNTD __m128i _mm_maskz_lzcnt_epi32(__mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_lzcnt_epi64(__m128i a);
VPLZCNTQ __m128i _mm_mask_lzcnt_epi64(__m128i s, __mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_maskz_lzcnt_epi64(__mmask8 m, __m128i a);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## VPMADD52HUQ—Packed Multiply of Unsigned 52-bit Unsigned Integers and Add High 52-bit Products to 64-bit Accumulators

Opcode/ Instruction	Op/ En	32/64 bit Mode Support	CPUID	Description
EVEX.DDS.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m(r)	NA

### Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

**Operation****VPMADD52HUQ (EVEX encoded)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64;

IF k1[j] OR \*no writemask\* THEN

IF src2 is Memory AND EVEX.b=1 THEN

tsrc2[63:0] ← ZeroExtend64(src2[51:0]);

ELSE

tsrc2[63:0] ← ZeroExtend64(src2[i+51:i]);

FI;

Temp128[127:0] ← ZeroExtend64(src1[i+51:i]) \* tsrc2[63:0];

Temp2[63:0] ← DEST[i+63:i] + ZeroExtend64(temp128[103:52]);

DEST[i+63:i] ← Temp2[63:0];

ELSE

IF \*zeroing-masking\* THEN

DEST[i+63:i] ← 0;

ELSE \*merge-masking\*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMADD52HUQ \_\_m512i \_\_mm512\_madd52hi\_epu64(\_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m512i \_\_mm512\_mask\_madd52hi\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m512i \_\_mm512\_maskz\_madd52hi\_epu64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_madd52hi\_epu64(\_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_mask\_madd52hi\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_maskz\_madd52hi\_epu64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m128i \_\_mm\_madd52hi\_epu64(\_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52HUQ \_\_m128i \_\_mm\_mask\_madd52hi\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52HUQ \_\_m128i \_\_mm\_maskz\_madd52hi\_epu64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPMADD52LUQ—Packed Multiply of Unsigned 52-bit Integers and Add the Low 52-bit Products to Qword Accumulators

Opcode/ Instruction	Op/En	32/64 bit Mode Support	CPUID	Description
EVEX.DDS.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 B4 /r VPMADD52LUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 B4 /r VPMADD52LUQ zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m(r)	NA

### Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.



**Operation****VPMADD52LUQ (EVEX encoded)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64;

IF k1[j] OR \*no writemask\* THEN

IF src2 is Memory AND EVEX.b=1 THEN

tsrc2[63:0] ← ZeroExtend64(src2[51:0]);

ELSE

tsrc2[63:0] ← ZeroExtend64(src2[i+51:i]);

FI;

Temp128[127:0] ← ZeroExtend64(src1[i+51:i]) \* tsrc2[63:0];

Temp2[63:0] ← DEST[i+63:i] + ZeroExtend64(temp128[51:0]);

DEST[i+63:i] ← Temp2[63:0];

ELSE

IF \*zeroing-masking\* THEN

DEST[i+63:i] ← 0;

ELSE \*merge-masking\*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMADD52LUQ \_\_m512i \_\_mm512\_madd52lo\_epu64( \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m512i \_\_mm512\_mask\_madd52lo\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m512i \_\_mm512\_maskz\_madd52lo\_epu64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_madd52lo\_epu64( \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_mask\_madd52lo\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_maskz\_madd52lo\_epu64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m128i \_\_mm\_madd52lo\_epu64( \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52LUQ \_\_m128i \_\_mm\_mask\_madd52lo\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52LUQ \_\_m128i \_\_mm\_maskz\_madd52lo\_epu64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPMASKMOV – Conditional SIMD Integer Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 8C /r VPMASKMOVD <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX2	Conditionally load dword values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 8C /r VPMASKMOVD <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX2	Conditionally load dword values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W1 8C /r VPMASKMOVQ <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX2	Conditionally load qword values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W1 8C /r VPMASKMOVQ <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX2	Conditionally load qword values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W0 8E /r VPMASKMOVD <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX2	Conditionally store dword values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W0 8E /r VPMASKMOVD <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX2	Conditionally store dword values from <i>ymm2</i> using mask in <i>ymm1</i> .
VEX.NDS.128.66.0F38.W1 8E /r VPMASKMOVQ <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX2	Conditionally store qword values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.NDS.256.66.0F38.W1 8E /r VPMASKMOVQ <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX2	Conditionally store qword values from <i>ymm2</i> using mask in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv	ModRM:reg (r)	NA

### Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are either XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VPMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm\_field, and the destination register is encoded in reg\_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg\_field, and the destination memory location is encoded in rm\_field.

## Operation

### VPMASKMOVD - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

### VPMASKMOVD -128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[MAXVL-1:128] ← 0
```

### VPMASKMOVQ - 256-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

### VPMASKMOVQ - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[MAXVL-1:128] ← 0
```

### VPMASKMOVD - 256-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ← SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
```

**VPMASKMOVD - 128-bit store**

IF (SRC1[31]) DEST[31:0]  $\leftarrow$  SRC2[31:0]  
 IF (SRC1[63]) DEST[63:32]  $\leftarrow$  SRC2[63:32]  
 IF (SRC1[95]) DEST[95:64]  $\leftarrow$  SRC2[95:64]  
 IF (SRC1[127]) DEST[127:96]  $\leftarrow$  SRC2[127:96]

**VPMASKMOVQ - 256-bit store**

IF (SRC1[63]) DEST[63:0]  $\leftarrow$  SRC2[63:0]  
 IF (SRC1[127]) DEST[127:64]  $\leftarrow$  SRC2[127:64]  
 IF (SRC1[191]) DEST[191:128]  $\leftarrow$  SRC2[191:128]  
 IF (SRC1[255]) DEST[255:192]  $\leftarrow$  SRC2[255:192]

**VPMASKMOVQ - 128-bit store**

IF (SRC1[63]) DEST[63:0]  $\leftarrow$  SRC2[63:0]  
 IF (SRC1[127]) DEST[127:64]  $\leftarrow$  SRC2[127:64]

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMASKMOVD: `__m256i _mm256_maskload_epi32(int const *a, __m256i mask)`  
 VPMASKMOVD: `void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)`  
 VPMASKMOVQ: `__m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);`  
 VPMASKMOVQ: `void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);`  
 VPMASKMOVD: `__m128i _mm_maskload_epi32(int const *a, __m128i mask)`  
 VPMASKMOVD: `void _mm_maskstore_epi32(int *a, __m128i mask, __m128 b)`  
 VPMASKMOVQ: `__m128i _mm_maskload_epi64(__int64 const *a, __m128i mask);`  
 VPMASKMOVQ: `void _mm_maskstore_epi64(__int64 *a, __m128i mask, __m128i b);`

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 6 (No AC# reported for any mask bit combinations).

## VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 29 /r VPMOVB2M k1, xmm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.
EVEX.256.F3.0F38.W0 29 /r VPMOVB2M k1, ymm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.
EVEX.512.F3.0F38.W0 29 /r VPMOVB2M k1, zmm1	RM	V/V	AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.
EVEX.128.F3.0F38.W1 29 /r VPMOVW2M k1, xmm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.
EVEX.256.F3.0F38.W1 29 /r VPMOVW2M k1, ymm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.
EVEX.512.F3.0F38.W1 29 /r VPMOVW2M k1, zmm1	RM	V/V	AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.
EVEX.128.F3.0F38.W0 39 /r VPMOVD2M k1, xmm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1.
EVEX.256.F3.0F38.W0 39 /r VPMOVD2M k1, ymm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1.
EVEX.512.F3.0F38.W0 39 /r VPMOVD2M k1, zmm1	RM	V/V	AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1.
EVEX.128.F3.0F38.W1 39 /r VPMOVQ2M k1, xmm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.
EVEX.256.F3.0F38.W1 39 /r VPMOVQ2M k1, ymm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.
EVEX.512.F3.0F38.W1 39 /r VPMOVQ2M k1, zmm1	RM	V/V	AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVB2M (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF SRC[i+7]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPMOVW2M (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF SRC[i+15]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPMOVD2M (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF SRC[i+31]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPMOVQ2M (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF SRC[i+63]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMPOVB2M __mmask64 _mm512_movepi8_mask( __m512i );
VPMPOVD2M __mmask16 _mm512_movepi32_mask( __m512i );
VPMPOVQ2M __mmask8 _mm512_movepi64_mask( __m512i );
VPMPOVW2M __mmask32 _mm512_movepi16_mask( __m512i );
VPMPOVB2M __mmask32 _mm256_movepi8_mask( __m256i );
VPMPOVD2M __mmask8 _mm256_movepi32_mask( __m256i );
VPMPOVQ2M __mmask8 _mm256_movepi64_mask( __m256i );
VPMPOVW2M __mmask16 _mm256_movepi16_mask( __m256i );
VPMPOVB2M __mmask16 _mm_movepi8_mask( __m128i );
VPMPOVD2M __mmask8 _mm_movepi32_mask( __m128i );
VPMPOVQ2M __mmask8 _mm_movepi64_mask( __m128i );
VPMPOVW2M __mmask8 _mm_movepi16_mask( __m128i );

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E7NM

#UD                      If EVEX.vvvv != 1111B.

## VPMOVDVB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 31 /r VPMOVDVB <i>xmm1/m32 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed double-word integers from <i>xmm2</i> into 4 packed byte integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 21 /r VPMOVSDB <i>xmm1/m32 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed double-word integers from <i>xmm2</i> into 4 packed signed byte integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 11 /r VPMOVUSDB <i>xmm1/m32 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned double-word integers from <i>xmm2</i> into 4 packed unsigned byte integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 31 /r VPMOVDVB <i>xmm1/m64 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed double-word integers from <i>ymm2</i> into 8 packed byte integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 21 /r VPMOVSDB <i>xmm1/m64 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed signed double-word integers from <i>ymm2</i> into 8 packed signed byte integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 11 /r VPMOVUSDB <i>xmm1/m64 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed unsigned double-word integers from <i>ymm2</i> into 8 packed unsigned byte integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 31 /r VPMOVDVB <i>xmm1/m128 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 16 packed double-word integers from <i>zmm2</i> into 16 packed byte integers in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 21 /r VPMOVSDB <i>xmm1/m128 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 16 packed signed double-word integers from <i>zmm2</i> into 16 packed signed byte integers in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB <i>xmm1/m128 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 16 packed unsigned double-word integers from <i>zmm2</i> into 16 packed unsigned byte integers in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVDVB down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMOVSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVUSDB convert unsigned double-word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.



**Operation****VPMOVDDB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateDoubleWordToByte (SRC[m+31:m])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;

```

**VPMOVDDB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

```

**VPMOVSDDB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedDoubleWordToByte (SRC[m+31:m])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;

```

**VPMOVSDB instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

m ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← SaturateSignedDoubleWordToByte (SRC[m+31:m])

ELSE \*DEST[i+7:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

**VPMOVUSDB instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

m ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← SaturateUnsignedDoubleWordToByte (SRC[m+31:m])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/4] ← 0;

**VPMOVUSDB instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

m ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← SaturateUnsignedDoubleWordToByte (SRC[m+31:m])

ELSE \*DEST[i+7:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPMOVB __m128i __mm512_cvtepi32_epi8( __m512i a);
VPMOVB __m128i __mm512_mask_cvtepi32_epi8( __m128i s, __mmask16 k, __m512i a);
VPMOVB __m128i __mm512_maskz_cvtepi32_epi8( __mmask16 k, __m512i a);
VPMOVB void __mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVSD __m128i __mm512_cvtsepi32_epi8( __m512i a);
VPMOVSD __m128i __mm512_mask_cvtsepi32_epi8( __m128i s, __mmask16 k, __m512i a);
VPMOVSD __m128i __mm512_maskz_cvtsepi32_epi8( __mmask16 k, __m512i a);
VPMOVSD void __mm512_mask_cvtsepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSD __m128i __mm512_cvtusepi32_epi8( __m512i a);
VPMOVUSD __m128i __mm512_mask_cvtusepi32_epi8( __m128i s, __mmask16 k, __m512i a);
VPMOVUSD __m128i __mm512_maskz_cvtusepi32_epi8( __mmask16 k, __m512i a);
VPMOVUSD void __mm512_mask_cvtusepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSD __m128i __mm256_cvtusepi32_epi8( __m256i a);
VPMOVUSD __m128i __mm256_mask_cvtusepi32_epi8( __m128i a, __mmask8 k, __m256i b);
VPMOVUSD __m128i __mm256_maskz_cvtusepi32_epi8( __mmask8 k, __m256i b);
VPMOVUSD void __mm256_mask_cvtusepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVUSD __m128i __mm_cvtusepi32_epi8( __m128i a);
VPMOVUSD __m128i __mm_mask_cvtusepi32_epi8( __m128i a, __mmask8 k, __m128i b);
VPMOVUSD __m128i __mm_maskz_cvtusepi32_epi8( __mmask8 k, __m128i b);
VPMOVUSD void __mm_mask_cvtusepi32_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVSD __m128i __mm256_cvtsepi32_epi8( __m256i a);
VPMOVSD __m128i __mm256_mask_cvtsepi32_epi8( __m128i a, __mmask8 k, __m256i b);
VPMOVSD __m128i __mm256_maskz_cvtsepi32_epi8( __mmask8 k, __m256i b);
VPMOVSD void __mm256_mask_cvtsepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVSD __m128i __mm_cvtsepi32_epi8( __m128i a);
VPMOVSD __m128i __mm_mask_cvtsepi32_epi8( __m128i a, __mmask8 k, __m128i b);
VPMOVSD __m128i __mm_maskz_cvtsepi32_epi8( __mmask8 k, __m128i b);
VPMOVSD void __mm_mask_cvtsepi32_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVB __m128i __mm256_cvtepi32_epi8( __m256i a);
VPMOVB __m128i __mm256_mask_cvtepi32_epi8( __m128i a, __mmask8 k, __m256i b);
VPMOVB __m128i __mm256_maskz_cvtepi32_epi8( __mmask8 k, __m256i b);
VPMOVB void __mm256_mask_cvtepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVB __m128i __mm_cvtepi32_epi8( __m128i a);
VPMOVB __m128i __mm_mask_cvtepi32_epi8( __m128i a, __mmask8 k, __m128i b);
VPMOVB __m128i __mm_maskz_cvtepi32_epi8( __mmask8 k, __m128i b);
VPMOVB void __mm_mask_cvtepi32_storeu_epi8(void *, __mmask8 k, __m128i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 33 /r VPMOVDW <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed double-word integers from <i>xmm2</i> into 4 packed word integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 23 /r VPMOVSDW <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed double-word integers from <i>xmm2</i> into 4 packed signed word integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 13 /r VPMOVUSDW <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned double-word integers from <i>xmm2</i> into 4 packed unsigned word integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 33 /r VPMOVDW <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed double-word integers from <i>ymm2</i> into 8 packed word integers in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 23 /r VPMOVSDW <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed signed double-word integers from <i>ymm2</i> into 8 packed signed word integers in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 13 /r VPMOVUSDW <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed unsigned double-word integers from <i>ymm2</i> into 8 packed unsigned word integers in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 33 /r VPMOVDW <i>ymm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 16 packed double-word integers from <i>zmm2</i> into 16 packed word integers in <i>ymm1/m256</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 23 /r VPMOVSDW <i>ymm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 16 packed signed double-word integers from <i>zmm2</i> into 16 packed signed word integers in <i>ymm1/m256</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW <i>ymm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 16 packed unsigned double-word integers from <i>zmm2</i> into 16 packed unsigned word integers in <i>ymm1/m256</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

```

**VPMOVDW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
        *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

```

**VPMOVSDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

```

**VPMOVSQ instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

m ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])

ELSE

\*DEST[i+15:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

**VPMOVSQ instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

m ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+15:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0;

**VPMOVSQ instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

m ← j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])

ELSE

\*DEST[i+15:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPMOVDW __m256i _mm512_cvtepi32_epi16(__m512i a);
VPMOVDW __m256i _mm512_mask_cvtepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void _mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_cvtsepi32_epi16(__m512i a);
VPMOVSDW __m256i _mm512_mask_cvtsepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
VPMOVSDW void _mm512_mask_cvtsepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_cvtusepi32_epi16(__m512i a);
VPMOVUSDW __m256i _mm512_mask_cvtusepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void _mm512_mask_cvtusepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m128i _mm256_cvtusepi32_epi16(__m256i a);
VPMOVUSDW __m128i _mm256_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm256_maskz_cvtusepi32_epi16(__mmask8 k, __m256i b);
VPMOVUSDW void _mm256_mask_cvtusepi32_storeu_epi16(void *, __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm_cvtusepi32_epi16(__m128i a);
VPMOVUSDW __m128i _mm_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDW __m128i _mm_maskz_cvtusepi32_epi16(__mmask8 k, __m128i b);
VPMOVUSDW void _mm_mask_cvtusepi32_storeu_epi16(void *, __mmask8 k, __m128i b);
VPMOVSDW __m128i _mm256_cvtsepi32_epi16(__m256i a);
VPMOVSDW __m128i _mm256_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVSDW __m128i _mm256_maskz_cvtsepi32_epi16(__mmask8 k, __m256i b);
VPMOVSDW void _mm256_mask_cvtsepi32_storeu_epi16(void *, __mmask8 k, __m256i b);
VPMOVSDW __m128i _mm_cvtsepi32_epi16(__m128i a);
VPMOVSDW __m128i _mm_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVSDW __m128i _mm_maskz_cvtsepi32_epi16(__mmask8 k, __m128i b);
VPMOVSDW void _mm_mask_cvtsepi32_storeu_epi16(void *, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm256_cvtepi32_epi16(__m256i a);
VPMOVDW __m128i _mm256_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_maskz_cvtepi32_epi16(__mmask8 k, __m256i b);
VPMOVDW void _mm256_mask_cvtepi32_storeu_epi16(void *, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm_cvtepi32_epi16(__m128i a);
VPMOVDW __m128i _mm_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm_maskz_cvtepi32_epi16(__mmask8 k, __m128i b);
VPMOVDW void _mm_mask_cvtepi32_storeu_epi16(void *, __mmask8 k, __m128i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1	RM	V/V	AVX512BW	Sets each byte in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 28 /r VPMOVM2W zmm1, k1	RM	V/V	AVX512BW	Sets each word in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1	RM	V/V	AVX512DQ	Sets each doubleword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 38 /r VPMOVM2Q zmm1, k1	RM	V/V	AVX512DQ	Sets each quadword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1's or all 0's depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.



**Operation****VPMOVM2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

i ← j \* 8

IF SRC[j]

THEN DEST[i+7:i] ← -1

ELSE DEST[i+7:i] ← 0

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPMOVM2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

i ← j \* 16

IF SRC[j]

THEN DEST[i+15:i] ← -1

ELSE DEST[i+15:i] ← 0

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPMOVM2D (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF SRC[j]

THEN DEST[i+31:i] ← -1

ELSE DEST[i+31:i] ← 0

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPMOVM2Q (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF SRC[j]

THEN DEST[i+63:i] ← -1

ELSE DEST[i+63:i] ← 0

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVM2B __m512i _mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i _mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i _mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i _mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i _mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i _mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i _mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i _mm256_movm_epi16(__mmask16);
VPMOVM2B __m128i _mm_movm_epi8(__mmask16);
VPMOVM2D __m128i _mm_movm_epi32(__mmask8);
VPMOVM2Q __m128i _mm_movm_epi64(__mmask8);
VPMOVM2W __m128i _mm_movm_epi16(__mmask8);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E7NM

#UD                      If EVEX.vvvv != 1111B.

## VPMOVBQ/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 32 /r VPMOVBQ <i>xmm1/m16 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed byte integers in <i>xmm1/m16</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 22 /r VPMOVSQB <i>xmm1/m16 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed signed quad-word integers from <i>xmm2</i> into 2 packed signed byte integers in <i>xmm1/m16</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB <i>xmm1/m16 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned byte integers in <i>xmm1/m16</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 32 /r VPMOVBQ <i>xmm1/m32 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed byte integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 22 /r VPMOVSQB <i>xmm1/m32 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed byte integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB <i>xmm1/m32 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned byte integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 32 /r VPMOVBQ <i>xmm1/m64 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed byte integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 22 /r VPMOVSQB <i>xmm1/m64 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed byte integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB <i>xmm1/m64 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned byte integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Eighth Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVBQ down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:64) of the destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/8] ← 0;

```

**VPMOVB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])
    ELSE
        *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

```

**VPMOVS instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+7:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/8] ← 0;

```

**VPMOVSQB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
    ELSE
        *DEST[i+7:i] remains unchanged*          ; merging-masking
    FI
ENDFOR

```

**VPMOVUSQB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
    ELSE
        IF *merging-masking*          ; merging-masking
            THEN *DEST[i+7:i] remains unchanged*
        ELSE *zeroing-masking*        ; zeroing-masking
            DEST[i+7:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL/8] ← 0;

```

**VPMOVUSQB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
    ELSE
        *DEST[i+7:i] remains unchanged*          ; merging-masking
    FI
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVBQ __m128i __mm512_cvtepi64_epi8( __m512i a);
VPMOVBQ __m128i __mm512_mask_cvtepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVBQ void __mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_cvtsepi64_epi8( __m512i a);
VPMOVBQ __m128i __mm512_mask_cvtsepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_maskz_cvtsepi64_epi8( __mmask8 k, __m512i a);
VPMOVBQ void __mm512_mask_cvtsepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_cvtusepi64_epi8( __m512i a);
VPMOVBQ __m128i __mm512_mask_cvtusepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_maskz_cvtusepi64_epi8( __mmask8 k, __m512i a);
VPMOVBQ void __mm512_mask_cvtusepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm256_cvtusepi64_epi8(__m256i a);
VPMOVBQ __m128i __mm256_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm256_maskz_cvtusepi64_epi8( __mmask8 k, __m256i b);
VPMOVBQ void __mm256_mask_cvtusepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm_cvtusepi64_epi8(__m128i a);
VPMOVBQ __m128i __mm_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm_maskz_cvtusepi64_epi8( __mmask8 k, __m128i b);
VPMOVBQ void __mm_mask_cvtusepi64_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm256_cvtsepi64_epi8(__m256i a);
VPMOVBQ __m128i __mm256_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm256_maskz_cvtsepi64_epi8( __mmask8 k, __m256i b);
VPMOVBQ void __mm256_mask_cvtsepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm_cvtsepi64_epi8(__m128i a);
VPMOVBQ __m128i __mm_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm_maskz_cvtsepi64_epi8( __mmask8 k, __m128i b);
VPMOVBQ void __mm_mask_cvtsepi64_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm256_cvtepi64_epi8(__m256i a);
VPMOVBQ __m128i __mm256_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm256_maskz_cvtepi64_epi8( __mmask8 k, __m256i b);
VPMOVBQ void __mm256_mask_cvtepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm_cvtepi64_epi8(__m128i a);
VPMOVBQ __m128i __mm_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm_maskz_cvtepi64_epi8( __mmask8 k, __m128i b);
VPMOVBQ void __mm_mask_cvtepi64_storeu_epi8(void *, __mmask8 k, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 35 /r VPMOVQD <i>xmm1/m128 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed double-word integers in <i>xmm1/m128</i> with truncation subject to writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 25 /r VPMOVSQD <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed signed quad-word integers from <i>xmm2</i> into 2 packed signed double-word integers in <i>xmm1/m64</i> using signed saturation subject to writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 15 /r VPMOVUSQD <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned double-word integers in <i>xmm1/m64</i> using unsigned saturation subject to writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 35 /r VPMOVQD <i>ymm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed double-word integers in <i>xmm1/m128</i> with truncation subject to writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 25 /r VPMOVSQD <i>ymm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed double-word integers in <i>xmm1/m128</i> using signed saturation subject to writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 15 /r VPMOVUSQD <i>ymm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned double-word integers in <i>xmm1/m128</i> using unsigned saturation subject to writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 35 /r VPMOVQD <i>zmm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed double-word integers in <i>ymm1/m256</i> with truncation subject to writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 25 /r VPMOVSQD <i>zmm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed double-word integers in <i>ymm1/m256</i> using signed saturation subject to writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 15 /r VPMOVUSQD <i>zmm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned double-word integers in <i>ymm1/m256</i> using unsigned saturation subject to writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVQD instruction (EVEX encoded version) reg-reg form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *zeroing-masking*          ; zeroing-masking
      DEST[i+31:i] ← 0
  FI
FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

```

**VPMOVQD instruction (EVEX encoded version) memory form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VPMOVSQD instruction (EVEX encoded version) reg-reg form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE
      IF *merging-masking*          ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking*        ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

```



**VPMOVSQD instruction (EVEX encoded version) memory form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR

```

**VPMOVUSQD instruction (EVEX encoded version) reg-reg form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*      ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking*    ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

```

**VPMOVUSQD instruction (EVEX encoded version) memory form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQD __m256i __mm512_cvtepi64_epi32( __m512i a);
VPMOVQD __m256i __mm512_mask_cvtepi64_epi32( __m256i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __mm512_maskz_cvtepi64_epi32( __mmask8 k, __m512i a);
VPMOVQD void __mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVSQD __m256i __mm512_cvtsepi64_epi32( __m512i a);
VPMOVSQD __m256i __mm512_mask_cvtsepi64_epi32( __m256i s, __mmask8 k, __m512i a);
VPMOVSQD __m256i __mm512_maskz_cvtsepi64_epi32( __mmask8 k, __m512i a);
VPMOVSQD void __mm512_mask_cvtsepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSD __m256i __mm512_cvtusepi64_epi32( __m512i a);
VPMOVUSD __m256i __mm512_mask_cvtusepi64_epi32( __m256i s, __mmask8 k, __m512i a);
VPMOVUSD __m256i __mm512_maskz_cvtusepi64_epi32( __mmask8 k, __m512i a);
VPMOVUSD void __mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSD __m128i __mm256_cvtusepi64_epi32( __m256i a);
VPMOVUSD __m128i __mm256_mask_cvtusepi64_epi32( __m128i a, __mmask8 k, __m256i b);
VPMOVUSD __m128i __mm256_maskz_cvtusepi64_epi32( __mmask8 k, __m256i b);
VPMOVUSD void __mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSD __m128i __mm_cvtusepi64_epi32( __m128i a);
VPMOVUSD __m128i __mm_mask_cvtusepi64_epi32( __m128i a, __mmask8 k, __m128i b);
VPMOVUSD __m128i __mm_maskz_cvtusepi64_epi32( __mmask8 k, __m128i b);
VPMOVUSD void __mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i __mm256_cvtsepi64_epi32( __m256i a);
VPMOVSQD __m128i __mm256_mask_cvtsepi64_epi32( __m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i __mm256_maskz_cvtsepi64_epi32( __mmask8 k, __m256i b);
VPMOVSQD void __mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i __mm_cvtsepi64_epi32( __m128i a);
VPMOVSQD __m128i __mm_mask_cvtsepi64_epi32( __m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i __mm_maskz_cvtsepi64_epi32( __mmask8 k, __m128i b);
VPMOVSQD void __mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i __mm256_cvtepi64_epi32( __m256i a);
VPMOVQD __m128i __mm256_mask_cvtepi64_epi32( __m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i __mm256_maskz_cvtepi64_epi32( __mmask8 k, __m256i b);
VPMOVQD void __mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i __mm_cvtepi64_epi32( __m128i a);
VPMOVQD __m128i __mm_mask_cvtepi64_epi32( __m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __mm_maskz_cvtepi64_epi32( __mmask8 k, __m128i b);
VPMOVQD void __mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 34 /r VPMOVQW <i>xmm1/m32</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed word integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 24 /r VPMOVSQW <i>xmm1/m32</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed signed quad-word integers from <i>xmm2</i> into 8 packed signed word integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 14 /r VPMOVUSQW <i>xmm1/m32</i> { <i>k1</i> } <i>{z}</i> , <i>xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned word integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 34 /r VPMOVQW <i>xmm1/m64</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed word integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 24 /r VPMOVSQW <i>xmm1/m64</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed word integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 14 /r VPMOVUSQW <i>xmm1/m64</i> { <i>k1</i> } <i>{z}</i> , <i>ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned word integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 34 /r VPMOVQW <i>xmm1/m128</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i>	A	V/V	AVX512F	Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed word integers in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 24 /r VPMOVSQW <i>xmm1/m128</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i>	A	V/V	AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed word integers in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 14 /r VPMOVUSQW <i>xmm1/m128</i> { <i>k1</i> } <i>{z}</i> , <i>zmm2</i>	A	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned word integers in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;

```

**VPMOVQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
        *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

```

**VPMOVSQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;

```

**VPMOVSQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
    ELSE
        *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR

```

**VPMOVUSQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking*              ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;

```

**VPMOVUSQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
        *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQW __m128i _mm512_cvtepi64_epi16(__m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_cvtsepi64_epi16(__m512i a);
VPMOVSQW __m128i _mm512_mask_cvtsepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_maskz_cvtsepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW void _mm512_mask_cvtsepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_cvtusepi64_epi16(__m512i a);
VPMOVUSQW __m128i _mm512_mask_cvtusepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_maskz_cvtusepi64_epi16(__mmask8 k, __m512i a);
VPMOVUSQW void _mm512_mask_cvtusepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void *, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32(__mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void *, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32(__mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void *, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32(__mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void *, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void *, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void *, __mmask8 k, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVBW/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 30 /r VPMOVBW <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512BW	Converts 8 packed word integers from <i>xmm2</i> into 8 packed bytes in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 20 /r VPMOVSWB <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512BW	Converts 8 packed signed word integers from <i>xmm2</i> into 8 packed signed bytes in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 10 /r VPMOVUSWB <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512BW	Converts 8 packed unsigned word integers from <i>xmm2</i> into 8 packed unsigned bytes in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 30 /r VPMOVBW <i>ymm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512BW	Converts 16 packed word integers from <i>ymm2</i> into 16 packed bytes in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 20 /r VPMOVSWB <i>ymm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512BW	Converts 16 packed signed word integers from <i>ymm2</i> into 16 packed signed bytes in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 10 /r VPMOVUSWB <i>ymm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512BW	Converts 16 packed unsigned word integers from <i>ymm2</i> into 16 packed unsigned bytes in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 30 /r VPMOVBW <i>zmm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512BW	Converts 32 packed word integers from <i>zmm2</i> into 32 packed bytes in <i>ymm1/m256</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 20 /r VPMOVSWB <i>zmm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512BW	Converts 32 packed signed word integers from <i>zmm2</i> into 32 packed signed bytes in <i>ymm1/m256</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 10 /r VPMOVUSWB <i>zmm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512BW	Converts 32 packed unsigned word integers from <i>zmm2</i> into 32 packed unsigned bytes in <i>ymm1/m256</i> using unsigned saturation under writemask <i>k1</i> .

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

### Description

VPMOVBW down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVB instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KI-1

i ← j \* 8

m ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0;

**VPMOVB instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KI-1

i ← j \* 8

m ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])

ELSE

\*DEST[i+7:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

**VPMOVSWB instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KI-1

i ← j \* 8

m ← j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] ← 0;



**VPMOVSWB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
  ELSE
    *DEST[i+7:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSWB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
  ELSE
    IF *merging-masking*                  ; merging-masking
    THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking*                ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

```

**VPMOVUSWB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
  ELSE
    *DEST[i+7:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVUSWB __m256i __mm512_cvtusepi16_epi8(__m512i a);
VPMOVUSWB __m256i __mm512_mask_cvtusepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB __m256i __mm512_maskz_cvtusepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSWB void __mm512_mask_cvtusepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVSWB __m256i __mm512_cvtsepi16_epi8(__m512i a);
VPMOVSWB __m256i __mm512_mask_cvtsepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVSWB __m256i __mm512_maskz_cvtsepi16_epi8(__mmask32 k, __m512i b);
VPMOVSWB void __mm512_mask_cvtsepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVWB __m256i __mm512_cvtepi16_epi8(__m512i a);
VPMOVWB __m256i __mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVWB __m256i __mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i b);
VPMOVWB void __mm512_mask_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVUSWB __m128i __mm256_cvtusepi16_epi8(__m256i a);
VPMOVUSWB __m128i __mm256_mask_cvtusepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i __mm256_maskz_cvtusepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void __mm256_mask_cvtusepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m128i __mm_cvtusepi16_epi8(__m128i a);
VPMOVUSWB __m128i __mm_mask_cvtusepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i __mm_maskz_cvtusepi16_epi8(__mmask8 k, __m128i b);
VPMOVUSWB void __mm_mask_cvtusepi16_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVSWB __m128i __mm256_cvtsepi16_epi8(__m256i a);
VPMOVSWB __m128i __mm256_mask_cvtsepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVSWB __m128i __mm256_maskz_cvtsepi16_epi8(__mmask16 k, __m256i b);
VPMOVSWB void __mm256_mask_cvtsepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVSWB __m128i __mm_cvtsepi16_epi8(__m128i a);
VPMOVSWB __m128i __mm_mask_cvtsepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSWB __m128i __mm_maskz_cvtsepi16_epi8(__mmask8 k, __m128i b);
VPMOVSWB void __mm_mask_cvtsepi16_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVWB __m128i __mm256_cvtepi16_epi8(__m256i a);
VPMOVWB __m128i __mm256_mask_cvtepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVWB __m128i __mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVWB void __mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVWB __m128i __mm_cvtepi16_epi8(__m128i a);
VPMOVWB __m128i __mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVWB __m128i __mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i b);
VPMOVWB void __mm_mask_cvtepi16_storeu_epi8(void *, __mmask8 k, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E6NF

#UD If EVEX.vvvv != 1111B.

## VPMULTISHIFTQB – Select Packed Unaligned Bytes from Quadword Sources

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 83 /r VPMULTISHIFTQB xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	AVX512_VBMI AVX512VL	Select unaligned bytes from qwords in xmm3/m128/m64bcst using control bytes in xmm2, write byte results to xmm1 under k1.
EVEX.NDS.256.66.0F38.W1 83 /r VPMULTISHIFTQB ymm1 {k1}{z}, ymm2,ymm3/m256/m64bcst	A	V/V	AVX512_VBMI AVX512VL	Select unaligned bytes from qwords in ymm3/m256/m64bcst using control bytes in ymm2, write byte results to ymm1 under k1.
EVEX.NDS.512.66.0F38.W1 83 /r VPMULTISHIFTQB zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512_VBMI	Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

Only the low 6 bits of each control byte are used to select an 8-bit slot to extract the output byte from the qword data in the second source operand. The starting bit of the 8-bit slot can be unaligned relative to any byte boundary and is left-shifted from the beginning of the input qword source by the amount specified in the low 6-bit of the control byte. If the 8-bit slot would exceed the qword boundary, the out-of-bound portion of the 8-bit slot is wrapped back to start from bit 0 of the input qword element.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register.

**Operation****VPMULTISHIFTQB DEST, SRC1, SRC2 (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR i ← 0 TO KL-1

IF EVEX.b=1 AND src2 is memory THEN

tcur ← src2.qword[0]; //broadcasting

ELSE

tcur ← src2.qword[i];

FI;

FOR j ← 0 to 7

ctrl ← src1.qword[i].byte[j] &amp; 63;

FOR k ← 0 to 7

res.bit[k] ← tcur.bit[ (ctrl+k) mod 64 ];

ENDFOR

IF k1[i\*8+j] or no writemask THEN

dst.qword[i].byte[j] ← res;

ELSE IF zeroing-masking THEN

dst.qword[i].byte[j] ← 0;

ENDFOR

ENDFOR

DEST.qword[MAX\_VL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULTISHIFTQB \_\_m512i \_mm512\_multishift\_epi64\_epi8( \_\_m512i a, \_\_m512i b);

VPMULTISHIFTQB \_\_m512i \_mm512\_mask\_multishift\_epi64\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMULTISHIFTQB \_\_m512i \_mm512\_maskz\_multishift\_epi64\_epi8( \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMULTISHIFTQB \_\_m256i \_mm256\_multishift\_epi64\_epi8( \_\_m256i a, \_\_m256i b);

VPMULTISHIFTQB \_\_m256i \_mm256\_mask\_multishift\_epi64\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMULTISHIFTQB \_\_m256i \_mm256\_maskz\_multishift\_epi64\_epi8( \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMULTISHIFTQB \_\_m128i \_mm\_multishift\_epi64\_epi8( \_\_m128i a, \_\_m128i b);

VPMULTISHIFTQB \_\_m128i \_mm\_mask\_multishift\_epi64\_epi8(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMULTISHIFTQB \_\_m128i \_mm\_maskz\_multishift\_epi64\_epi8( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.

**VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 15 /r VPROLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.
EVEX.NDD.128.66.0F.W0 72 /1 ib VPROLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 15 /r VPROLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.
EVEX.NDD.128.66.0F.W1 72 /1 ib VPROLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 15 /r VPROLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.
EVEX.NDD.256.66.0F.W0 72 /1 ib VPROLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 15 /r VPROLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.
EVEX.NDD.256.66.0F.W1 72 /1 ib VPROLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 15 /r VPROLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /1 ib VPROLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512F	Rotate left of doublewords in zmm2/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 15 /r VPROLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1 under writemask k1.
EVEX.NDD.512.66.0F.W1 72 /1 ib VPROLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512F	Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

**Operation**

```
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC << COUNT) | (SRC >> (32 - COUNT));
```

```
LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC << COUNT) | (SRC >> (64 - COUNT));
```

**VPROLD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

    i ← j \* 32

    IF k1[j] OR \*no writemask\* THEN

        IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

            THEN DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[31:0], imm8)

            ELSE DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[i+31:i], imm8)

        FI;

    ELSE

        IF \*merging-masking\* ; merging-masking

            THEN \*DEST[i+31:i] remains unchanged\*

        ELSE \*zeroing-masking\* ; zeroing-masking

            DEST[i+31:i] ← 0

        FI

    FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPROLVD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
      ELSE DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPROLQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:i] ← LEFT_ROTATE_QWORDS(SRC1[63:0], imm8)
      ELSE DEST[i+63:i] ← LEFT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPROLVQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[63:0])

ELSE DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPROLD \_\_m512i \_mm512\_rol\_epi32(\_\_m512i a, int imm);

VPROLD \_\_m512i \_mm512\_mask\_rol\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, int imm);

VPROLD \_\_m512i \_mm512\_maskz\_rol\_epi32(\_\_mmask16 k, \_\_m512i a, int imm);

VPROLD \_\_m256i \_mm256\_rol\_epi32(\_\_m256i a, int imm);

VPROLD \_\_m256i \_mm256\_mask\_rol\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);

VPROLD \_\_m256i \_mm256\_maskz\_rol\_epi32(\_\_mmask8 k, \_\_m256i a, int imm);

VPROLD \_\_m128i \_mm\_rol\_epi32(\_\_m128i a, int imm);

VPROLD \_\_m128i \_mm\_mask\_rol\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);

VPROLD \_\_m128i \_mm\_maskz\_rol\_epi32(\_\_mmask8 k, \_\_m128i a, int imm);

VPROLQ \_\_m512i \_mm512\_rol\_epi64(\_\_m512i a, int imm);

VPROLQ \_\_m512i \_mm512\_mask\_rol\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, int imm);

VPROLQ \_\_m512i \_mm512\_maskz\_rol\_epi64(\_\_mmask8 k, \_\_m512i a, int imm);

VPROLQ \_\_m256i \_mm256\_rol\_epi64(\_\_m256i a, int imm);

VPROLQ \_\_m256i \_mm256\_mask\_rol\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);

VPROLQ \_\_m256i \_mm256\_maskz\_rol\_epi64(\_\_mmask8 k, \_\_m256i a, int imm);

VPROLQ \_\_m128i \_mm\_rol\_epi64(\_\_m128i a, int imm);

VPROLQ \_\_m128i \_mm\_mask\_rol\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);

VPROLQ \_\_m128i \_mm\_maskz\_rol\_epi64(\_\_mmask8 k, \_\_m128i a, int imm);

VPROLVD \_\_m512i \_mm512\_rolv\_epi32(\_\_m512i a, \_\_m512i cnt);

VPROLVD \_\_m512i \_mm512\_mask\_rolv\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, \_\_m512i cnt);

VPROLVD \_\_m512i \_mm512\_maskz\_rolv\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);

VPROLVD \_\_m256i \_mm256\_rolv\_epi32(\_\_m256i a, \_\_m256i cnt);

VPROLVD \_\_m256i \_mm256\_mask\_rolv\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);

VPROLVD \_\_m256i \_mm256\_maskz\_rolv\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);

VPROLVD \_\_m128i \_mm\_rolv\_epi32(\_\_m128i a, \_\_m128i cnt);

VPROLVD \_\_m128i \_mm\_mask\_rolv\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);

VPROLVD \_\_m128i \_mm\_maskz\_rolv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

VPROLVQ \_\_m512i \_mm512\_rolv\_epi64(\_\_m512i a, \_\_m512i cnt);

VPROLVQ \_\_m512i \_mm512\_mask\_rolv\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, \_\_m512i cnt);

VPROLVQ \_\_m512i \_mm512\_maskz\_rolv\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);

VPROLVQ \_\_m256i \_mm256\_rolv\_epi64(\_\_m256i a, \_\_m256i cnt);

VPROLVQ \_\_m256i \_mm256\_mask\_rolv\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);

VPROLVQ \_\_m256i \_mm256\_maskz\_rolv\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);

VPROLVQ \_\_m128i \_mm\_rolv\_epi64(\_\_m128i a, \_\_m128i cnt);



VPROLVQ \_\_m128i \_\_mm\_mask\_rolv\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
VPROLVQ \_\_m128i \_\_mm\_maskz\_rolv\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

**VPRORD/VPRORVD/VPRORQ/VPRORVQ—Bit Rotate Right**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 14 /r VPRORVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /0 ib VPRORD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.
EVEX.NDS.128.66.0F38.W1 14 /r VPRORVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.
EVEX.NDD.128.66.0F.W1 72 /0 ib VPRORQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.
EVEX.NDS.256.66.0F38.W0 14 /r VPRORVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store using result writemask k1.
EVEX.NDD.256.66.0F.W0 72 /0 ib VPRORD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.
EVEX.NDS.256.66.0F38.W1 14 /r VPRORVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.
EVEX.NDD.256.66.0F.W1 72 /0 ib VPRORQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.
EVEX.NDS.512.66.0F38.W0 14 /r VPRORVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /0 ib VPRORD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512F	Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.
EVEX.NDS.512.66.0F38.W1 14 /r VPRORVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.
EVEX.NDD.512.66.0F.W1 72 /0 ib VPRORQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512F	Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

## Operation

```
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC >> COUNT) | (SRC << (32 - COUNT));
```

```
RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC >> COUNT) | (SRC << (64 - COUNT));
```

### VPRORD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] ← RIGHT_ROTATE_DWORDS( SRC1[31:0], imm8)
      ELSE DEST[i+31:i] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

**VPRORVD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[31:0])

ELSE DEST[i+31:i] ← RIGHT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPRORQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[63:0], imm8)

ELSE DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[i+63:i], imm8)

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPRORVQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[63:0])

ELSE DEST[i+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPRORD \_\_m512i \_mm512\_ror\_epi32(\_\_m512i a, int imm);

VPRORD \_\_m512i \_mm512\_mask\_ror\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, int imm);

VPRORD \_\_m512i \_mm512\_maskz\_ror\_epi32(\_\_mmask16 k, \_\_m512i a, int imm);

VPRORD \_\_m256i \_mm256\_ror\_epi32(\_\_m256i a, int imm);

VPRORD \_\_m256i \_mm256\_mask\_ror\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);

VPRORD \_\_m256i \_mm256\_maskz\_ror\_epi32(\_\_mmask8 k, \_\_m256i a, int imm);

VPRORD \_\_m128i \_mm\_ror\_epi32(\_\_m128i a, int imm);

VPRORD \_\_m128i \_mm\_mask\_ror\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);

VPRORD \_\_m128i \_mm\_maskz\_ror\_epi32(\_\_mmask8 k, \_\_m128i a, int imm);

VPRORQ \_\_m512i \_mm512\_ror\_epi64(\_\_m512i a, int imm);

VPRORQ \_\_m512i \_mm512\_mask\_ror\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, int imm);

VPRORQ \_\_m512i \_mm512\_maskz\_ror\_epi64(\_\_mmask8 k, \_\_m512i a, int imm);

VPRORQ \_\_m256i \_mm256\_ror\_epi64(\_\_m256i a, int imm);

VPRORQ \_\_m256i \_mm256\_mask\_ror\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, int imm);

VPRORQ \_\_m256i \_mm256\_maskz\_ror\_epi64(\_\_mmask8 k, \_\_m256i a, int imm);

VPRORQ \_\_m128i \_mm\_ror\_epi64(\_\_m128i a, int imm);

VPRORQ \_\_m128i \_mm\_mask\_ror\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, int imm);

VPRORQ \_\_m128i \_mm\_maskz\_ror\_epi64(\_\_mmask8 k, \_\_m128i a, int imm);

VPRORVD \_\_m512i \_mm512\_rorv\_epi32(\_\_m512i a, \_\_m512i cnt);

VPRORVD \_\_m512i \_mm512\_mask\_rorv\_epi32(\_\_m512i a, \_\_mmask16 k, \_\_m512i b, \_\_m512i cnt);

VPRORVD \_\_m512i \_mm512\_maskz\_rorv\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);

VPRORVD \_\_m256i \_mm256\_rorv\_epi32(\_\_m256i a, \_\_m256i cnt);

VPRORVD \_\_m256i \_mm256\_mask\_rorv\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);

VPRORVD \_\_m256i \_mm256\_maskz\_rorv\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);

VPRORVD \_\_m128i \_mm\_rorv\_epi32(\_\_m128i a, \_\_m128i cnt);

VPRORVD \_\_m128i \_mm\_mask\_rorv\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);

VPRORVD \_\_m128i \_mm\_maskz\_rorv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

VPRORVQ \_\_m512i \_mm512\_rorv\_epi64(\_\_m512i a, \_\_m512i cnt);

VPRORVQ \_\_m512i \_mm512\_mask\_rorv\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i b, \_\_m512i cnt);

VPRORVQ \_\_m512i \_mm512\_maskz\_rorv\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);

VPRORVQ \_\_m256i \_mm256\_rorv\_epi64(\_\_m256i a, \_\_m256i cnt);

VPRORVQ \_\_m256i \_mm256\_mask\_rorv\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i b, \_\_m256i cnt);

VPRORVQ \_\_m256i \_mm256\_maskz\_rorv\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);

VPRORVQ \_\_m128i \_mm\_rorv\_epi64(\_\_m128i a, \_\_m128i cnt);

VPRORVQ \_\_m128i \_\_mm\_mask\_rorv\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i b, \_\_m128i cnt);  
VPRORVQ \_\_m128i \_\_mm\_maskz\_rorv\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);

#### SIMD Floating-Point Exceptions

None

#### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

## VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A0 /vsib VPSCATTERDD vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A0 /vsib VPSCATTERDD vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A0 /vsib VPSCATTERDD vm32z {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.128.66.0F38.W0 A1 /vsib VPSCATTERQD vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A1 /vsib VPSCATTERQD vm64y {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A1 /vsib VPSCATTERQD vm64z {k1}, ymm1	A	V/V	AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64z {k1}, zmm1	A	V/V	AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

### Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address BASE\_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.

- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules.  $N$  is considered to be the size of a single vector element. The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

The instruction will #UD fault if EVEX.Z = 1.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VPSCATTERDD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

    IF  $k1[j]$  OR \*no writemask\*

        THEN  $\text{MEM}[\text{BASE\_ADDR} + \text{SignExtend}(\text{VINDEX}[i+31:i]) * \text{SCALE} + \text{DISP}] \leftarrow \text{SRC}[i+31:i]$

$k1[j] \leftarrow 0$

    FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$

#### VPSCATTERDQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

$k \leftarrow j * 32$

    IF  $k1[j]$  OR \*no writemask\*

        THEN  $\text{MEM}[\text{BASE\_ADDR} + \text{SignExtend}(\text{VINDEX}[k+31:k]) * \text{SCALE} + \text{DISP}] \leftarrow \text{SRC}[i+63:i]$

$k1[j] \leftarrow 0$

    FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$



**VPSCATTERQD (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + (VINDEK[k+63:k]) \* SCALE + DISP] ← SRC[i+31:i]

k1[j] ← 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**VPSCATTERQQ (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN MEM[BASE\_ADDR + (VINDEK[j+63:j]) \* SCALE + DISP] ← SRC[i+63:i]

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSCATTERDD void \_\_mm512\_i32scatter\_epi32(void \* base, \_\_m512i vdx, \_\_m512i a, int scale);

VPSCATTERDD void \_\_mm256\_i32scatter\_epi32(void \* base, \_\_m256i vdx, \_\_m256i a, int scale);

VPSCATTERDD void \_\_mm\_i32scatter\_epi32(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERDD void \_\_mm512\_mask\_i32scatter\_epi32(void \* base, \_\_mmask16 k, \_\_m512i vdx, \_\_m512i a, int scale);

VPSCATTERDD void \_\_mm256\_mask\_i32scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m256i a, int scale);

VPSCATTERDD void \_\_mm\_mask\_i32scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERDQ void \_\_mm512\_i32scatter\_epi64(void \* base, \_\_m512i vdx, \_\_m512i a, int scale);

VPSCATTERDQ void \_\_mm256\_i32scatter\_epi64(void \* base, \_\_m256i vdx, \_\_m256i a, int scale);

VPSCATTERDQ void \_\_mm\_i32scatter\_epi64(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERDQ void \_\_mm512\_mask\_i32scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m512i vdx, \_\_m512i a, int scale);

VPSCATTERDQ void \_\_mm256\_mask\_i32scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m256i a, int scale);

VPSCATTERDQ void \_\_mm\_mask\_i32scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERQD void \_\_mm512\_i64scatter\_epi32(void \* base, \_\_m512i vdx, \_\_m256i a, int scale);

VPSCATTERQD void \_\_mm256\_i64scatter\_epi32(void \* base, \_\_m256i vdx, \_\_m128i a, int scale);

VPSCATTERQD void \_\_mm\_i64scatter\_epi32(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERQD void \_\_mm512\_mask\_i64scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m512i vdx, \_\_m256i a, int scale);

VPSCATTERQD void \_\_mm256\_mask\_i64scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m128i a, int scale);

VPSCATTERQD void \_\_mm\_mask\_i64scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERQQ void \_\_mm512\_i64scatter\_epi64(void \* base, \_\_m512i vdx, \_\_m512i a, int scale);

VPSCATTERQQ void \_\_mm256\_i64scatter\_epi64(void \* base, \_\_m256i vdx, \_\_m256i a, int scale);

VPSCATTERQQ void \_\_mm\_i64scatter\_epi64(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERQQ void \_\_mm512\_mask\_i64scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m512i vdx, \_\_m512i a, int scale);

VPSCATTERQQ void \_\_mm256\_mask\_i64scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m256i a, int scale);

VPSCATTERQQ void \_\_mm\_mask\_i64scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

**VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 47 /r VPSLLVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 47 /r VPSLLVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.NDS.128.66.0F38.W1 12 /r VPSLLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	AVX512VL AVX512BW	Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 12 /r VPSLLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	AVX512VL AVX512BW	Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 12 /r VPSLLVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW	Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W0 47 /r VPSLLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W0 47 /r VPSLLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W0 47 /r VPSLLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 47 /r VPSLLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation****VPSLLVW (EVEX encoded version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
  ELSE
    IF *merging-masking*                ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE                                ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;

```

**VPSLLVD (VEX.128 version)**

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
IF COUNT_0 < 32 THEN
  DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
  DEST[31:0] ← 0;
  (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
  DEST[127:96] ← ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
  DEST[127:96] ← 0;
DEST[MAXVL-1:128] ← 0;

```

**VPSLLVD (VEX.256 version)**

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 224];
IF COUNT_0 < 32 THEN
  DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
  DEST[31:0] ← 0;
  (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
  DEST[255:224] ← ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
  DEST[255:224] ← 0;
DEST[MAXVL-1:256] ← 0;

```

**VPSLLVD (EVEX encoded version)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
      ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;

```

**VPSLLVQ (VEX.128 version)**

```

COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
    DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
    DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
    DEST[127:64] ← ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
    DEST[127:96] ← 0;
DEST[MAXVL-1:128] ← 0;

```

**VPSLLVQ (VEX.256 version)**

```

COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[197 : 192];
IF COUNT_0 < 64 THEN
    DEST[63:0] ← ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
    DEST[63:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
    DEST[255:192] ← ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
    DEST[255:192] ← 0;
DEST[MAXVL-1:256] ← 0;

```

**VPSLLVQ (EVEX encoded version)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
            ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSLLVW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m512i _mm512_sllv_epi32(__m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_mask_sllv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_maskz_sllv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m512i _mm512_sllv_epi64(__m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_mask_sllv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_maskz_sllv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m256i _mm256_sllv_epi32 (__m256i m, __m256i count)
VPSLLVQ __m256i _mm256_sllv_epi64 (__m256i m, __m256i count)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 4.  
 EVEX-encoded VPSLLVD/VPSLLVQ, see Exceptions Type E4.  
 EVEX-encoded VPSLLVW, see Exceptions Type E4.nb.

**VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 46 /r VPSRAVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.
VEX.NDS.256.66.0F38.W0 46 /r VPSRAVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.
EVEX.NDS.128.66.0F38.W1 11 /r VPSRAVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F38.W1 11 /r VPSRAVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W1 11 /r VPSRAVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F38.W0 46 /r VPSRAVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F38.W0 46 /r VPSRAVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W0 46 /r VPSRAVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F38.W1 46 /r VPSRAVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F38.W1 46 /r VPSRAVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W1 46 /r VPSRAVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA



## Description

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element are filled with the corresponding sign bit of the source element.

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 16 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

## Operation

### VPSRAVW (EVEX encoded version)

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      COUNT ← SRC2[i+3:i]
      IF COUNT < 16
        THEN DEST[i+15:i] ← SignExtend(SRC1[i+15:i] >> COUNT)
        ELSE
          FOR k ← 0 TO 15
            DEST[i+k] ← SRC1[i+15]
          ENDFOR;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+15:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+15:i] ← 0
          FI
        FI;
      ENDFOR;
    DEST[MAXVL-1:VL] ← 0;

```

**VPSRAVD (VEX.128 version)**

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] ← 0;

```

**VPSRAVD (VEX.256 version)**

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 224];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAXVL-1:256] ← 0;

```

**VPSRAVD (EVEX encoded version)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        COUNT ← SRC2[4:0]
        IF COUNT < 32
          THEN DEST[i+31:i] ← SignExtend(SRC1[i+31:i] >> COUNT)
          ELSE
            FOR k ← 0 TO 31
              DEST[i+k] ← SRC1[i+31]
            ENDFOR;
          FI
        ELSE
          COUNT ← SRC2[i+4:i]
          IF COUNT < 32
            THEN DEST[i+31:i] ← SignExtend(SRC1[i+31:i] >> COUNT)
            ELSE
              FOR k ← 0 TO 31
                DEST[i+k] ← SRC1[i+31]
              ENDFOR;
            FI
          FI;
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
          ELSE ; zeroing-masking
            DEST[31:0] ← 0
          FI
        FI;
      ENDFOR;
    DEST[MAXVL-1:VL] ← 0;

```

**VPSRAVQ (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

COUNT ← SRC2[5:0]

IF COUNT &lt; 64

THEN DEST[i+63:i] ← SignExtend(SRC1[i+63:i] &gt;&gt; COUNT)

ELSE

FOR k ← 0 TO 63

DEST[i+k] ← SRC1[i+63]

ENDFOR;

FI

ELSE

COUNT ← SRC2[i+5:i]

IF COUNT &lt; 64

THEN DEST[i+63:i] ← SignExtend(SRC1[i+63:i] &gt;&gt; COUNT)

ELSE

FOR k ← 0 TO 63

DEST[i+k] ← SRC1[i+63]

ENDFOR;

FI

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

DEST[63:0] ← 0

FI

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m128i _mm_srav_epi32(__m128i a, __m128i cnt);
VPSRAVD __m128i _mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m512i _mm512_srav_epi64(__m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_mask_srav_epi64(__m512i s, __mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_maskz_srav_epi64(__mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m256i _mm256_srav_epi64(__m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_maskz_srav_epi64(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m128i _mm_srav_epi64(__m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_maskz_srav_epi64(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m512i _mm512_srav_epi16(__m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_mask_srav_epi16(__m512i s, __mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_maskz_srav_epi16(__mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m256i _mm256_srav_epi16(__m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_mask_srav_epi16(__m256i s, __mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_maskz_srav_epi16(__mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m128i _mm_srav_epi16(__m128i a, __m128i cnt);
VPSRAVW __m128i _mm_mask_srav_epi16(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i m, __m256i count)

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

**VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.NDS.128.66.0F38.W1 10 /r VPSRLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 10 /r VPSRLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 10 /r VPSRLVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W0 45 /r VPSRLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W0 45 /r VPSRLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W0 45 /r VPSRLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 45 /r VPSRLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSRLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSRLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation****VPSRLVW (EVEX encoded version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← ZeroExtend(SRC1[i+15:i] >> SRC2[i+15:i])
    ELSE
      IF *merging-masking*                ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE                                ; zeroing-masking
        DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;

```

**VPSRLVD (VEX.128 version)**

```

COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96];
IF COUNT_0 < 32 THEN
  DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
  DEST[31:0] ← 0;
  (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
  DEST[127:96] ← ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
  DEST[127:96] ← 0;
DEST[MAXVL-1:128] ← 0;

```

**VPSRLVD (VEX.256 version)**

```

COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255 : 224];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
    DEST[31:0] ← 0;
    (* Repeat shift operation for 2nd through 7th dwords *)
    IF COUNT_7 < 32 THEN
        DEST[255:224] ← ZeroExtend(SRC1[255:224] >> COUNT_7);
    ELSE
        DEST[255:224] ← 0;
    DEST[MAXVL-1:256] ← 0;

```

**VPSRLVD (EVEX encoded version)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
            ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    ENDFOR;
    DEST[MAXVL-1:VL] ← 0;

```

**VPSRLVQ (VEX.128 version)**

```

COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
    DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
    DEST[63:0] ← 0;
    IF COUNT_1 < 64 THEN
        DEST[127:64] ← ZeroExtend(SRC1[127:64] >> COUNT_1);
    ELSE
        DEST[127:64] ← 0;
    DEST[MAXVL-1:128] ← 0;

```

**VPSRLVQ (VEX.256 version)**

```

COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] ← ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
DEST[255:192] ← 0;
DEST[MAXVL-1:256] ← 0;

```

**VPSRLVQ (EVEX encoded version)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
      ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;

```



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSRLVW __m512i _mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_maskz_srlv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_mask_srlv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m256i _mm256_maskz_srlv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m128i _mm_mask_srlv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m128i _mm_maskz_srlv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m256i _mm256_srlv_epi32(__m256i m, __m256i count);
VPSRLVD __m512i _mm512_srlv_epi32(__m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_mask_srlv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_maskz_srlv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m256i _mm256_mask_srlv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m256i _mm256_maskz_srlv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m128i _mm_mask_srlv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVD __m128i _mm_maskz_srlv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m512i _mm512_srlv_epi64(__m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_mask_srlv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_maskz_srlv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m256i _mm256_mask_srlv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m256i _mm256_maskz_srlv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m128i _mm_mask_srlv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_maskz_srlv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m256i _mm256_srlv_epi64(__m256i m, __m256i count);
VPSRLVD __m128i _mm_srlv_epi32(__m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_srlv_epi64(__m128i a, __m128i cnt);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 4.  
 EVEX-encoded VPSRLVD/Q, see Exceptions Type E4.  
 EVEX-encoded VPSRLVW, see Exceptions Type E4.nb.

## VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F3A.W0 25 /r ib VPTERNLOGD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m32bcst as source operands and writing the result to xmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.256.66.0F3A.W0 25 /r ib VPTERNLOGD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m32bcst as source operands and writing the result to ymm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.512.66.0F3A.W0 25 /r ib VPTERNLOGD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m32bcst as source operands and writing the result to zmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.128.66.0F3A.W1 25 /r ib VPTERNLOGQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m64bcst as source operands and writing the result to xmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.256.66.0F3A.W1 25 /r ib VPTERNLOGQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m64bcst as source operands and writing the result to ymm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.512.66.0F3A.W1 25 /r ib VPTERNLOGQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m64bcst as source operands and writing the result to zmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

Table 5-18 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

**Table 5-18. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values**

VPTERNLOGD reg1, reg2, src3, 0xE2			Bit Result with Imm8=0xE2	VPTERNLOGD reg1, reg2, src3, 0xE4			Bit Result with Imm8=0xE4
Bit(reg1)	Bit(reg2)	Bit(src3)		Bit(reg1)	Bit(reg2)	Bit(src3)	
0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	1
0	1	1	0	0	1	1	0
1	0	0	0	1	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-10 and Table 5-11 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

### Operation

#### VPTERNLOGD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      FOR k ← 0 TO 31

        IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

          THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[k]]

          ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]

        FI;

      ; table lookup of immediate bellow;

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[31+i:i] remains unchanged\*

      ELSE ; zeroing-masking

        DEST[31+i:i] ← 0

    FI;

  FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**VPTERNLOGQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

FOR k ← 0 TO 63

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[j][k] ← imm[(DEST[i+k] &lt;&lt; 2) + (SRC1[i+k] &lt;&lt; 1) + SRC2[k]]

ELSE DEST[j][k] ← imm[(DEST[i+k] &lt;&lt; 2) + (SRC1[i+k] &lt;&lt; 1) + SRC2[i+k]]

FI; ; table lookup of immediate bellow;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63+i:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[63+i:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPTERNLOGD \_\_m512i \_\_mm512\_ternarylogic\_epi32(\_\_m512i a, \_\_m512i b, int imm);

VPTERNLOGD \_\_m512i \_\_mm512\_mask\_ternarylogic\_epi32(\_\_m512i s, \_\_mmask16 m, \_\_m512i a, \_\_m512i b, int imm);

VPTERNLOGD \_\_m512i \_\_mm512\_maskz\_ternarylogic\_epi32(\_\_mmask m, \_\_m512i a, \_\_m512i b, int imm);

VPTERNLOGD \_\_m256i \_\_mm256\_ternarylogic\_epi32(\_\_m256i a, \_\_m256i b, int imm);

VPTERNLOGD \_\_m256i \_\_mm256\_mask\_ternarylogic\_epi32(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i b, int imm);

VPTERNLOGD \_\_m256i \_\_mm256\_maskz\_ternarylogic\_epi32(\_\_mmask8 m, \_\_m256i a, \_\_m256i b, int imm);

VPTERNLOGD \_\_m128i \_\_mm\_ternarylogic\_epi32(\_\_m128i a, \_\_m128i b, int imm);

VPTERNLOGD \_\_m128i \_\_mm\_mask\_ternarylogic\_epi32(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b, int imm);

VPTERNLOGD \_\_m128i \_\_mm\_maskz\_ternarylogic\_epi32(\_\_mmask8 m, \_\_m128i a, \_\_m128i b, int imm);

VPTERNLOGQ \_\_m512i \_\_mm512\_ternarylogic\_epi64(\_\_m512i a, \_\_m512i b, int imm);

VPTERNLOGQ \_\_m512i \_\_mm512\_mask\_ternarylogic\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a, \_\_m512i b, int imm);

VPTERNLOGQ \_\_m512i \_\_mm512\_maskz\_ternarylogic\_epi64(\_\_mmask8 m, \_\_m512i a, \_\_m512i b, int imm);

VPTERNLOGQ \_\_m256i \_\_mm256\_ternarylogic\_epi64(\_\_m256i a, \_\_m256i b, int imm);

VPTERNLOGQ \_\_m256i \_\_mm256\_mask\_ternarylogic\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i b, int imm);

VPTERNLOGQ \_\_m256i \_\_mm256\_maskz\_ternarylogic\_epi64(\_\_mmask8 m, \_\_m256i a, \_\_m256i b, int imm);

VPTERNLOGQ \_\_m128i \_\_mm\_ternarylogic\_epi64(\_\_m128i a, \_\_m128i b, int imm);

VPTERNLOGQ \_\_m128i \_\_mm\_mask\_ternarylogic\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b, int imm);

VPTERNLOGQ \_\_m128i \_\_mm\_maskz\_ternarylogic\_epi64(\_\_mmask8 m, \_\_m128i a, \_\_m128i b, int imm);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 26 /r VPTESTMB k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise AND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W0 26 /r VPTESTMB k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise AND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W0 26 /r VPTESTMB k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512BW	Bitwise AND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.66.0F38.W1 26 /r VPTESTMW k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise AND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W1 26 /r VPTESTMW k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise AND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W1 26 /r VPTESTMW k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512BW	Bitwise AND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.66.0F38.W0 27 /r VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W0 27 /r VPTESTMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W0 27 /r VPTESTMD k2 {k1}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

**VPTESTMD/VPTESTMQ:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

**VPTESTMB/VPTESTMW:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

**Operation****VPTESTMB (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;
    ELSE DEST[j] = 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**VPTESTMW (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;
    ELSE DEST[j] = 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**VPTESTMD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;
        ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;
      FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

**VPTESTMQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;

ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;

FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPTESTMB \_\_mmask64 \_mm512\_test\_epi8\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMB \_\_mmask64 \_mm512\_mask\_test\_epi8\_mask(\_\_mmask64, \_\_m512i a, \_\_m512i b);

VPTESTMW \_\_mmask32 \_mm512\_test\_epi16\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMW \_\_mmask32 \_mm512\_mask\_test\_epi16\_mask(\_\_mmask32, \_\_m512i a, \_\_m512i b);

VPTESTMD \_\_mmask16 \_mm512\_test\_epi32\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMD \_\_mmask16 \_mm512\_mask\_test\_epi32\_mask(\_\_mmask16, \_\_m512i a, \_\_m512i b);

VPTESTMQ \_\_mmask8 \_mm512\_test\_epi64\_mask(\_\_m512i a, \_\_m512i b);

VPTESTMQ \_\_mmask8 \_mm512\_mask\_test\_epi64\_mask(\_\_mmask8, \_\_m512i a, \_\_m512i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VPTESTMD/Q: See Exceptions Type E4.

VPTESTMB/W: See Exceptions Type E4.nb.

## VPTESTNMB/W/D/Q—Logical NAND and Set

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID	Description
EVEX.NDS.128.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512F AVX512BW	Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512F AVX512BW	Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.



## Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

## Operation

**VPTESTNMB**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1

    i ← j\*8

    IF MaskBit(j) OR \*no writemask\*

        THEN

            DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0

        ELSE DEST[j] ← 0; zeroing masking only

    FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTNMW**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1

    i ← j\*16

    IF MaskBit(j) OR \*no writemask\*

        THEN

            DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0

        ELSE DEST[j] ← 0; zeroing masking only

    FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTNMD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j\*32

IF MaskBit(j) OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0

ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0

FI

ELSE DEST[j] ← 0; zeroing masking only

FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTNMQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j\*64

IF MaskBit(j) OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;

ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;

FI;

ELSE DEST[j] ← 0; zeroing masking only

FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPTESTNMB \_\_mmask64 \_\_mm512\_testn\_epi8\_mask( \_\_m512i a, \_\_m512i b);

VPTESTNMB \_\_mmask64 \_\_mm512\_mask\_testn\_epi8\_mask(\_\_mmask64, \_\_m512i a, \_\_m512i b);

VPTESTNMB \_\_mmask32 \_\_mm256\_testn\_epi8\_mask(\_\_m256i a, \_\_m256i b);

VPTESTNMB \_\_mmask32 \_\_mm256\_mask\_testn\_epi8\_mask(\_\_mmask32, \_\_m256i a, \_\_m256i b);

VPTESTNMB \_\_mmask16 \_\_mm\_testn\_epi8\_mask(\_\_m128i a, \_\_m128i b);

VPTESTNMB \_\_mmask16 \_\_mm\_mask\_testn\_epi8\_mask(\_\_mmask16, \_\_m128i a, \_\_m128i b);

VPTESTNMW \_\_mmask32 \_\_mm512\_testn\_epi16\_mask( \_\_m512i a, \_\_m512i b);

VPTESTNMW \_\_mmask32 \_\_mm512\_mask\_testn\_epi16\_mask(\_\_mmask32, \_\_m512i a, \_\_m512i b);

VPTESTNMW \_\_mmask16 \_\_mm256\_testn\_epi16\_mask(\_\_m256i a, \_\_m256i b);

VPTESTNMW \_\_mmask16 \_\_mm256\_mask\_testn\_epi16\_mask(\_\_mmask16, \_\_m256i a, \_\_m256i b);

VPTESTNMW \_\_mmask8 \_\_mm\_testn\_epi16\_mask(\_\_m128i a, \_\_m128i b);

VPTESTNMW \_\_mmask8 \_\_mm\_mask\_testn\_epi16\_mask(\_\_mmask8, \_\_m128i a, \_\_m128i b);

VPTESTNMD \_\_mmask16 \_\_mm512\_testn\_epi32\_mask( \_\_m512i a, \_\_m512i b);

VPTESTNMD \_\_mmask16 \_\_mm512\_mask\_testn\_epi32\_mask(\_\_mmask16, \_\_m512i a, \_\_m512i b);

VPTESTNMD \_\_mmask8 \_\_mm256\_testn\_epi32\_mask(\_\_m256i a, \_\_m256i b);

VPTESTNMD \_\_mmask8 \_\_mm256\_mask\_testn\_epi32\_mask(\_\_mmask8, \_\_m256i a, \_\_m256i b);

VPTESTNMD \_\_mmask8 \_\_mm\_testn\_epi32\_mask(\_\_m128i a, \_\_m128i b);

VPTESTNMD \_\_mmask8 \_\_mm\_mask\_testn\_epi32\_mask(\_\_mmask8, \_\_m128i a, \_\_m128i b);

VPTESTNMQ \_\_mmask8 \_\_mm512\_testn\_epi64\_mask(\_\_m512i a, \_\_m512i b);

VPTESTNMQ \_\_mmask8 \_\_mm512\_mask\_testn\_epi64\_mask(\_\_mmask8, \_\_m512i a, \_\_m512i b);

VPTESTNMQ \_\_mmask8 \_\_mm256\_testn\_epi64\_mask(\_\_m256i a, \_\_m256i b);

VPTESTNMQ \_\_mmask8 \_\_mm256\_mask\_testn\_epi64\_mask(\_\_mmask8, \_\_m256i a, \_\_m256i b);

VPTESTNMQ \_\_mmask8 \_\_mm\_testn\_epi64\_mask(\_\_m128i a, \_\_m128i b);

VPTESTNMQ \_\_mmask8 \_mm\_mask\_testn\_epi64\_mask(\_\_mmask8, \_\_m128i a, \_\_m128i b);

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

VPTESTNMD/VPTESTNMQ: See Exceptions Type E4.

VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb.

## VRANGEPD—Range Restriction Calculation For Packed Pairs of Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 50 /r ib VRANGEPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Calculate two RANGE operation output value from 2 pairs of double-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.256.66.0F3A.W1 50 /r ib VRANGEPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Calculate four RANGE operation output value from 4pairs of double-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.512.66.0F3A.W1 50 /r ib VRANGEPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	A	V/V	AVX512DQ	Calculate eight RANGE operation output value from 8 pairs of double-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

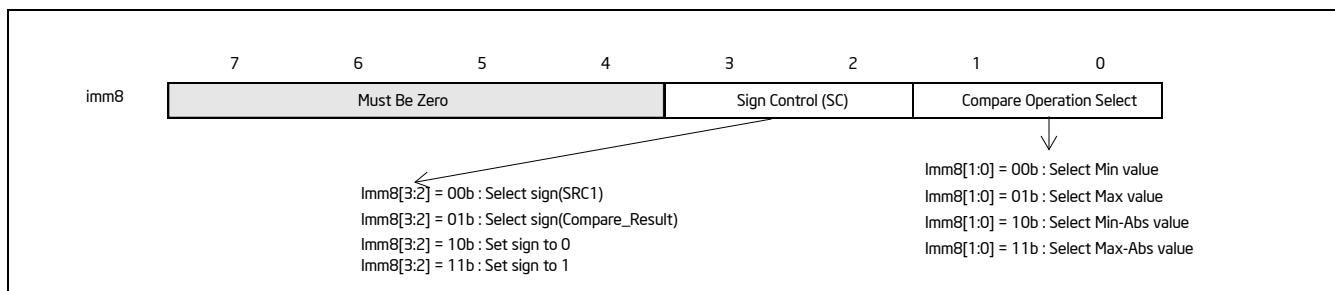


Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS

When one or more of the input value is a NaN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NaN is listed in Table 5-19. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-19.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGE PD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-20.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-21.

**Table 5-19. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]**

Src1	Src2	Result	IE Signaling Due to Comparison	Imm8[3:2] Effect to Range Output
sNaN1	sNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	qNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	Norm2	Quiet(sNaN1)	Yes	Ignored
qNaN1	sNaN2	Quiet(sNaN2)	Yes	Ignored
qNaN1	qNaN2	qNaN1	No	Applicable
qNaN1	Norm2	Norm2	No	Applicable
Norm1	sNaN2	Quiet(sNaN2)	Yes	Ignored
Norm1	qNaN2	Norm1	No	Applicable

**Table 5-20. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN\_ABS and MAX, MAX\_ABS**

MIN and MIN_ABS			MAX and MAX_ABS		
Src1	Src2	Result	Src1	Src2	Result
+0	-0	-0	+0	-0	+0
-0	+0	-0	-0	+0	+0

**Table 5-21. Comparison Result of Equal-Magnitude Input Cases for MIN\_ABS and MAX\_ABS, ( $|a| = |b|$ ,  $a > 0$ ,  $b < 0$ )**

MIN_ABS ( $ a  =  b $ , $a > 0$ , $b < 0$ )			MAX_ABS ( $ a  =  b $ , $a > 0$ , $b < 0$ )		
Src1	Src2	Result	Src1	Src2	Result
a	b	b	a	b	a
b	a	b	b	a	a

**Operation**

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```
{
    // Check if SNAN and report IE, see also Table 5-19
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[62:52];
    Src1.fraction ← SRC1[51:0];
    IF ((Src1.exp = 0) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[62:52];
    Src2.fraction ← SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction != 0)) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
    ELSE IF (SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-20
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-21
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest ← TMP[63:0];// Preserve sign of compare result
    10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}
```

CmpOpCtl[1:0]= imm8[1:0];  
SignSelCtl[1:0]=imm8[3:2];

**VRANGE PD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);

ELSE DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] = 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 1023$ .

```
VRANGE PD zmm_dst, zmm_src, zmm_1023, 02h;
```

Where:

zmm\_dst is the destination operand.

zmm\_src is the input operand to compare against  $\pm 1023$  (this is SRC1).

zmm\_1023 is the reference operand, contains the value of 1023 (and this is SRC2).

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.

In case  $|zmm\_src| < 1023$  (i.e. SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm\_dst. Otherwise, the value stored in zmm\_dst will get the value of 1023 (received on zmm\_1023, which is SRC2).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm\_src. So, even in the case of  $|zmm\_src| \geq 1023$ , the selected sign of SRC1 is kept.

Thus, if  $zmm\_src < -1023$ , the result of VRANGE PD will be the minimal value of -1023 while if  $zmm\_src > +1023$ , the result of VRANGE PD will be the maximal value of +1023.

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRANGEPD __m512d __mm512_range_pd ( __m512d a, __m512d b, int imm);
VRANGEPD __m512d __mm512_range_round_pd ( __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d __mm512_mask_range_pd ( __m512 ds, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d __mm512_mask_range_round_pd ( __m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d __mm512_maskz_range_pd ( __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d __mm512_maskz_range_round_pd ( __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d __mm256_range_pd ( __m256d a, __m256d b, int imm);
VRANGEPD __m256d __mm256_mask_range_pd ( __m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d __mm256_maskz_range_pd ( __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d __mm_range_pd ( __m128d a, __m128d b, int imm);
VRANGEPD __m128d __mm_mask_range_pd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d __mm_maskz_range_pd ( __mmask8 k, __m128d a, __m128d b, int imm);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E2.



## VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 50 /r ib VRANGEPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.256.66.0F3A.W0 50 /r ib VRANGEPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.512.66.0F3A.W0 50 /r ib VRANGEPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	A	V/V	AVX512DQ	Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-19. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-19.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-20.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-21.

**Operation**

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```
{
    // Check if SNAN and report IE, see also Table 5-19
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[30:23];
    Src1.fraction ← SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp ← SRC2[30:23];
    Src2.fraction ← SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-20
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-21
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest ← TMP[31:0];// Preserve sign of compare result
    10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}
```

CmpOpCtl[1:0]= imm8[1:0];

SignSelCtl[1:0]=imm8[3:2];

**VRANGEPS**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);

ELSE DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[i+31:i], CmpOpCtl[1:0], SignSelCtl[1:0]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] = 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 150$ .

```
VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;
```

Where:

zmm\_dst is the destination operand.

zmm\_src is the input operand to compare against  $\pm 150$ .

zmm\_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|zmm\_src| < 150$ , then its value will be written into zmm\_dst. Otherwise, the value stored in zmm\_dst will get the value of 150 (received on zmm\_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm\_src. So, even in the case of  $|zmm\_src| \geq 150$ , the selected sign of SRC1 is kept.

Thus, if  $zmm\_src < -150$ , the result of VRANGEPS will be the minimal value of -150 while if  $zmm\_src > +150$ , the result of VRANGE will be the maximal value of +150.

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRANGEPS __m512_mm512_range_ps ( __m512 a, __m512 b, int imm);
VRANGEPS __m512_mm512_range_round_ps ( __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512_mm512_mask_range_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512_mm512_mask_range_round_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512_mm512_maskz_range_ps ( __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512_mm512_maskz_range_round_ps ( __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256_mm256_range_ps ( __m256 a, __m256 b, int imm);
VRANGEPS __m256_mm256_mask_range_ps ( __m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256_mm256_maskz_range_ps ( __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128_mm_range_ps ( __m128 a, __m128 b, int imm);
VRANGEPS __m128_mm_mask_range_ps ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128_mm_maskz_range_ps ( __mmask8 k, __m128 a, __m128 b, int imm);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E2.

## VRANGESD—Range Restriction Calculation From a pair of Scalar Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 51 /r VRANGESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512DQ	Calculate a RANGE operation output value from 2 double-precision floating-point values in xmm2 and xmm3/m64, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

This instruction calculates a range operation output from two input double-precision FP values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-19. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-19.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-20.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-21.

**Operation**

```

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-19
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[62:52];
    Src1.fraction ← SRC1[51:0];
    IF ((Src1.exp = 0) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[62:52];
    Src2.fraction ← SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction != 0)) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
    ELSE IF (SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-20
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-21
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest ← TMP[63:0];// Preserve sign of compare result
    10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0] = imm8[1:0];
SignSelCtl[1:0] = imm8[3:2];

```

**VRANGESD**

```

IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[63:0] = 0
FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 1023$ .

```
VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;
```

Where:

xmm\_dst is the destination operand.

xmm\_src is the input operand to compare against  $\pm 1023$ .

xmm\_1023 is the reference operand, contains the value of 1023.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|xmm\_src| < 1023$ , then its value will be written into xmm\_dst. Otherwise, the value stored in xmm\_dst will get the value of 1023 (received on xmm\_1023).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm\_src. So, even in the case of  $|xmm\_src| \geq 1023$ , the selected sign of SRC1 is kept.

Thus, if  $xmm\_src < -1023$ , the result of VRANGESD will be the minimal value of -1023 while if  $xmm\_src > +1023$ , the result of VRANGESD will be the maximal value of +1023.

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRANGESD __m128d __mm_range_sd ( __m128d a, __m128d b, int imm);
VRANGESD __m128d __mm_range_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d __mm_mask_range_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d __mm_mask_range_round_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d __mm_maskz_range_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d __mm_maskz_range_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E3.

## VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 51 /r VRANGESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ	Calculate a RANGE operation output value from 2 single-precision floating-point values in xmm2 and xmm3/m32, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction calculates a range operation output from two input single-precision FP values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

Bits 128:31 of the destination operand are copied from the respective elements of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-19. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-19.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-20.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-21.



**Operation**

```

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-19
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[30:23];
    Src1.fraction ← SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp ← SRC2[30:23];
    Src2.fraction ← SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-20
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-21
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest ← TMP[31:0];// Preserve sign of compare result
    10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

**VRANGESS**

```

IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[31:0] = 0
    FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 150$ .

```
VRANGESS zmm_dst, zmm_src, zmm_150, 02h;
```

Where:

zmm\_dst is the destination operand.

zmm\_src is the input operand to compare against  $\pm 150$ .

zmm\_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|zmm\_src| < 150$ , then its value will be written into zmm\_dst. Otherwise, the value stored in zmm\_dst will get the value of 150 (received on zmm\_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm\_src. So, even in the case of  $|zmm\_src| \geq 150$ , the selected sign of SRC1 is kept.

Thus, if zmm\_src < -150, the result of VRANGESS will be the minimal value of -150 while if zmm\_src > +150, the result of VRANGE will be the maximal value of +150.

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRANGESS __m128 __mm_range_ss ( __m128 a, __m128 b, int imm);
VRANGESS __m128 __mm_range_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 __mm_mask_range_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 __mm_mask_range_round_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 __mm_maskz_range_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 __mm_maskz_range_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E3.

## VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4C /r VRCP14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4C /r VRCP14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4C /r VRCP14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand. The maximum relative error for this approximation is less than  $2^{-14}$ .

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-22. VRCP14PD/VRCP14SD Special Cases**

Input value	Result value	Comments
$0 \leq X \leq 2^{-1024}$	INF	Very small denormal
$-2^{-1024} \leq X \leq -0$	-INF	Very small denormal
$X > 2^{1022}$	Underflow	Up to 18 bits of fractions are returned*
$X < -2^{1022}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	$2^n$	
$X = -2^{-n}$	$-2^n$	

\* in this case the mantissa is shifted right by one or two bits

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

**Operation****VRCP14PD ((EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+63:i] ← APPROXIMATE(1.0/SRC[63:0]);
      ELSE DEST[i+63:i] ← APPROXIMATE(1.0/SRC[i+63:i]);
    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[i+63:i] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PD \_\_m512d \_mm512\_rcp14\_pd( \_\_m512d a);

VRCP14PD \_\_m512d \_mm512\_mask\_rcp14\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VRCP14PD \_\_m512d \_mm512\_maskz\_rcp14\_pd( \_\_mmask8 k, \_\_m512d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 4D /r VRCP14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F	Computes the approximate reciprocal of the scalar double-precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double-precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-22 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP14SD (EVEX version)

```

IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← APPROXIMATE(1.0/SRC2[63:0]);
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

`VRCP14SD __m128d _mm_rcp14_sd( __m128d a, __m128d b);`

`VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);`

`VRCP14SD __m128d _mm_maskz_rcp14_sd( __mmask8 k, __m128d a, __m128d b);`

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E5.

## VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4C /r VRCP14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4C /r VRCP14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4C /r VRCP14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than  $2^{-14}$ .

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-23. VRCP14PS/VRCP14SS Special Cases**

Input value	Result value	Comments
$0 \leq X \leq 2^{-128}$	INF	Very small denormal
$-2^{-128} \leq X \leq -0$	-INF	Very small denormal
$X > 2^{126}$	Underflow	Up to 18 bits of fractions are returned*
$X < -2^{126}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	$2^n$	
$X = -2^{-n}$	$-2^n$	

\* in this case the mantissa is shifted right by one or two bits

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vcrt14-vrsqrt14-vcrt28-vrsqrt28-vexp2>.

**Operation****VRCP14PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← APPROXIMATE(1.0/SRC[31:0]);

ELSE DEST[i+31:i] ← APPROXIMATE(1.0/SRC[i+31:i]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PS \_\_m512 \_\_mm512\_rcp14\_ps( \_\_m512 a);

VRCP14PS \_\_m512 \_\_mm512\_mask\_rcp14\_ps( \_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VRCP14PS \_\_m512 \_\_mm512\_maskz\_rcp14\_ps( \_\_mmask16 k, \_\_m512 a);

VRCP14PS \_\_m256 \_\_mm256\_rcp14\_ps( \_\_m256 a);

VRCP14PS \_\_m256 \_\_mm512\_mask\_rcp14\_ps( \_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VRCP14PS \_\_m256 \_\_mm512\_maskz\_rcp14\_ps( \_\_mmask8 k, \_\_m256 a);

VRCP14PS \_\_m128 \_\_mm\_rcp14\_ps( \_\_m128 a);

VRCP14PS \_\_m128 \_\_mm\_mask\_rcp14\_ps( \_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VRCP14PS \_\_m128 \_\_mm\_maskz\_rcp14\_ps( \_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.



## VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 4D /r VRCP14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F	Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-23 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP14SS (EVEX version)

```

IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← APPROXIMATE(1.0/SRC2[31:0]);
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[31:0] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[31:0] ← 0
    FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

`VRCP14SS __m128_mm_rcp14_ss( __m128 a, __m128 b);`

`VRCP14SS __m128_mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);`

`VRCP14SS __m128_mm_maskz_rcp14_ss( __mmask8 k, __m128 a, __m128 b);`

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E5.

## VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 56 /r ib VREDUCEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed double-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W1 56 /r ib VREDUCEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed double-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W1 56 /r ib VREDUCEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512DQ	Perform reduction transformation on double-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Perform reduction transformation of the packed binary encoded double-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where “Round()” treats “src”, “ $2^M$ ”, and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man2}$ , where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE:  $0 <= |\text{Reduced Result}| <= 2^{p-M-1}$

Then if RC  $\neq$  RNE:  $0 <= |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

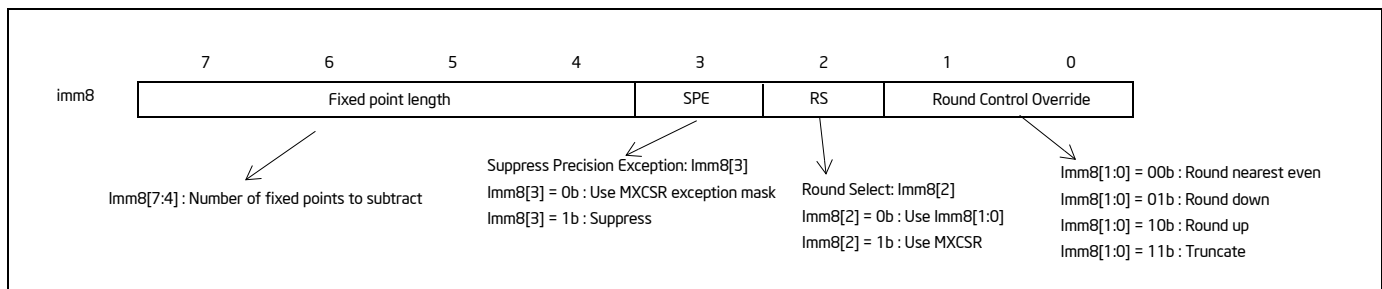


Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-24.

**Table 5-24. VREDUCEPD/SD/PS/SS Special Cases**

	Round Mode	Returned value
$ Src1  < 2^{-M-1}$	RNE	Src1
$ Src1  < 2^{-M}$	RPI, $Src1 > 0$	Round $(Src1 - 2^{-M})$ *
	RPI, $Src1 \leq 0$	Src1
	RNI, $Src1 \geq 0$	Src1
	RNI, $Src1 < 0$	Round $(Src1 + 2^{-M})$ *
Src1 = $\pm 0$ , or Dest = $\pm 0$ (Src1 != INF)	NOT RNI	+0.0
	RNI	-0.0
Src1 = $\pm INF$	any	+0.0
Src1 = $\pm NAN$	n/a	QNaN(Src1)

\* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

### Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0]; // Round Control for ROUND() operation
    RC_source ← imm[2];
    SPE ← 0; // Suppress Precision Exception
    TMP[63:0] ←  $2^{-M} * \{ \text{ROUND}(2^M * \text{SRC}[63:0], \text{SPE}, \text{RC\_source}, \text{RC}) \}$ ; // ROUND() treats SRC and  $2^M$  as standard binary FP values
    TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

### VREDUCEPD

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b == 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i] ← ReduceArgumentDP(SRC[63:0], imm8[7:0]);

      ELSE DEST[i+63:i] ← ReduceArgumentDP(SRC[i+63:i], imm8[7:0]);

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] = 0

    FI;

  FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VREDUCEPD __m512d_mm512_mask_reduce_pd( __m512d a, int imm, int sae)
VREDUCEPD __m512d_mm512_mask_reduce_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m512d_mm512_maskz_reduce_pd(__mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m256d_mm256_mask_reduce_pd( __m256d a, int imm)
VREDUCEPD __m256d_mm256_mask_reduce_pd(__m256d s, __mmask8 k, __m256d a, int imm)
VREDUCEPD __m256d_mm256_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
VREDUCEPD __m128d_mm_mask_reduce_pd( __m128d a, int imm)
VREDUCEPD __m128d_mm_mask_reduce_pd(__m128d s, __mmask8 k, __m128d a, int imm)
VREDUCEPD __m128d_mm_maskz_reduce_pd(__mmask8 k, __m128d a, int imm)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2, additionally

#UD                      If EVEX.vvvv != 1111B.

## VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 57 VREDUCESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8/r	A	V/V	AVX512D Q	Perform a reduction transformation on a scalar double-precision floating point value in xmm3/m64 by subtracting a number of fraction bits specified by the imm8 field. Also, upper double precision floating-point value (bits[127:64]) from xmm2 are copied to xmm1[127:64]. Stores the result in xmm1 register.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Perform a reduction transformation of the binary encoded double-precision FP value in the low qword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where “Round()” treats “src”, “ $2^M$ ”, and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man2}$ , where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC  $\neq$  RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 5-24.

### Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0]; // Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0; // Suppress Precision Exception
    TMP[63:0] ←  $2^{-M} * \text{ROUND}(2^M * \text{SRC}[63:0], \text{SPE}, \text{RC\_source}, \text{RC})$ ; // ROUND() treats SRC and  $2^M$  as standard binary FP values
    TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

**VREDUCESD**

```

IF k1[0] or *no writemask*
  THEN  DEST[63:0] ← ReduceArgumentDP(SRC2[63:0], imm8[7:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                           ; zeroing-masking
      THEN DEST[63:0] = 0
    FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VREDUCESD __m128d _mm_mask_reduce_sd( __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 56 /r ib VREDUCEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed single-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W0 56 /r ib VREDUCEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed single-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W0 56 /r ib VREDUCEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512DQ	Perform reduction transformation on packed single-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Perform reduction transformation of the packed binary encoded single-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where “Round()” treats “src”, “ $2^M$ ”, and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man2}$ , where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC  $\neq$  RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-24.



**Operation**

```

ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NaN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0]; // Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0; // Suppress Precision Exception
    TMP[31:0] ← 2-M * {ROUND(2M * SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
    TMP[31:0] ← SRC[31:0] - TMP[31:0]; // subtraction under the same RC, SPE controls
    RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}

```

**VREDUCEPS**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b == 1) AND (SRC *is memory*)
            THEN DEST[i+31:i] ← ReduceArgumentSP(SRC[31:0], imm8[7:0]);
            ELSE DEST[i+31:i] ← ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VREDUCEPS __m512 __mm512_mask_reduce_ps( __m512 a, int imm, int sae)
VREDUCEPS __m512 __mm512_mask_reduce_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m512 __mm512_maskz_reduce_ps(__mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m256 __mm256_mask_reduce_ps( __m256 a, int imm)
VREDUCEPS __m256 __mm256_mask_reduce_ps(__m256 s, __mmask8 k, __m256 a, int imm)
VREDUCEPS __m256 __mm256_maskz_reduce_ps(__mmask8 k, __m256 a, int imm)
VREDUCEPS __m128 __mm_mask_reduce_ps( __m128 a, int imm)
VREDUCEPS __m128 __mm_mask_reduce_ps(__m128 s, __mmask8 k, __m128 a, int imm)
VREDUCEPS __m128 __mm_maskz_reduce_ps(__mmask8 k, __m128 a, int imm)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2, additionally

#UD If EVEX.vvvv != 1111B.

## VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 57 /r /ib VREDUCESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ	Perform a reduction transformation on a scalar single-precision floating point value in xmm3/m32 by subtracting a number of fraction bits specified by the imm8 field. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. Stores the result in xmm1 register.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Perform a reduction transformation of the binary encoded single-precision FP value in the low dword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where “Round()” treats “src”, “ $2^M$ ”, and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man2}$ , where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC  $\neq$  RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-24.

### Operation

ReduceArgumentSP(SRC[31:0], imm8[7:0])

```
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0]; // Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0; // Suppress Precision Exception
    TMP[31:0] ←  $2^{-M} * \{\text{ROUND}(2^M * \text{SRC}[31:0], \text{SPE}, \text{RC\_source}, \text{RC})\}$ ; // ROUND() treats SRC and  $2^M$  as standard binary FP values
    TMP[31:0] ← SRC[31:0] - TMP[31:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

**VREDUCESS**

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] ← ReduceArgumentSP(SRC2[31:0], imm8[7:0])
  ELSE
    IF *merging-masking*                ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                                ; zeroing-masking
      THEN DEST[31:0] = 0
    FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VREDUCESS __m128 __mm_mask_reduce_ss( __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 __mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 __mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

**VRNDSCALEPD—Round Packed Float64 Values To Include A Given Number Of Fraction Bits**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 09 /r ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed double-precision floating point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W1 09 /r ib VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed double-precision floating point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W1 09 /r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F	Rounds packed double-precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

**Description**

Round the double-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM/YMM/XMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPD is

$$\text{ROUND}(x) = 2^{-M} \cdot \text{Round\_to\_INT}(x \cdot 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x \cdot 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

```
ROUND(x) = Round_to_INT(x, round_ctrl),
round_ctrl = imm[3:0];
```

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

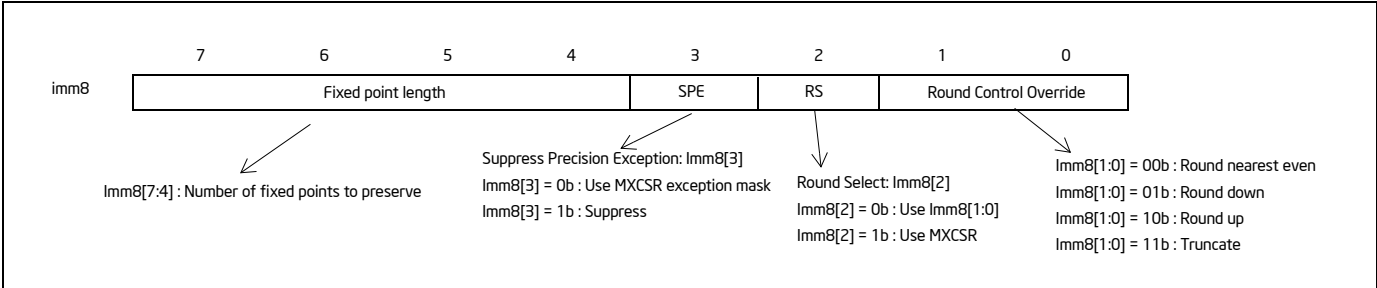


Figure 5-29. Imm8 Controls for VRNDSCALEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-25.

Table 5-25. VRNDSCALEPD/SD/PS/SS Special Cases

	Returned value
Src1=±inf	Src1
Src1=±NAN	Src1 converted to QNAN
Src1=±0	Src1

**Operation**

```

RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC      ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]      ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]                          ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(2M*SRC[63:0])
    01: TMP[63:0] ← round_to_equal_or_smaller_integer(2M*SRC[63:0])
    10: TMP[63:0] ← round_to_equal_or_larger_integer(2M*SRC[63:0])
    11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[63:0])
    ESAC

    Dest[63:0] ← 2-M* TMP[63:0]             ; scale down back to 2-M

    if (imm8[3] = 0) Then ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}

```

**VRNDSCALEPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF \*src is a memory operand\*

THEN TMP\_SRC ← BROADCAST64(SRC, VL, k1)

ELSE TMP\_SRC ← SRC

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← RoundToIntegerDP((TMP\_SRC[i+63:i], imm8[7:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALEPD __m512d _mm512_roundscale_pd( __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_roundscale_round_pd( __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_mask_roundscale_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_mask_roundscale_round_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_pd( __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_round_pd( __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_roundscale_pd( __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_mask_roundscale_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_pd( __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m128d _mm_roundscale_pd( __m128d a, int imm);
VRNDSCALEPD __m128d _mm_mask_roundscale_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm_maskz_roundscale_pd( __mmask8 k, __m128d a, int imm);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2.

**VRNDSCALESD—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits**

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 0B /r ib VRNDSCALESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F	Rounds scalar double-precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register.

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

**Description**

Rounds a double-precision floating-point value in the low quadword (see Figure 5-29) element the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the third operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESD is

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-25.



**Operation**

```

RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR.RC      ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]      ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]                          ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(2M*SRC[63:0])
    01: TMP[63:0] ← round_to_equal_or_smaller_integer(2M*SRC[63:0])
    10: TMP[63:0] ← round_to_equal_or_larger_integer(2M*SRC[63:0])
    11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[63:0])
    ESAC

    Dest[63:0] ← 2-M* TMP[63:0]             ; scale down back to 2-M

    if (imm8[3] = 0) Then ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}

```

VRNDSCALESD (EVEX encoded version)

```

IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundToIntegerDP(SRC2[63:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
    FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALESD __m128d __mm_roundscale_sd ( __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d __mm_roundscale_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d __mm_mask_roundscale_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d __mm_mask_roundscale_round_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d __mm_maskz_roundscale_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d __mm_maskz_roundscale_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 08 /r ib VRNDSCALEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed single-precision floating point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W0 08 /r ib VRNDSCALEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed single-precision floating point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W0 08 /r ib VRNDSCALEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F	Rounds packed single-precision floating-point values in zmm2/m512/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

### Description

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

$$\begin{aligned}\text{ROUND}(x) &= 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \\ M &= \text{imm}[7:4];\end{aligned}$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

$$\begin{aligned}\text{ROUND}(x) &= \text{Round\_to\_INT}(x, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0];\end{aligned}$$

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.  
Handling of special case of input values are listed in Table 5-25.

### Operation

```

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR.RC      ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]      ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]                          ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] ← round_to_nearest_even_integer(2M*SRC[31:0])
    01: TMP[31:0] ← round_to_equal_or_smaller_integer(2M*SRC[31:0])
    10: TMP[31:0] ← round_to_equal_or_larger_integer(2M*SRC[31:0])
    11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[31:0])
    ESAC;

    Dest[31:0] ← 2-M* TMP[31:0]             ; scale down back to 2-M
    if (imm8[3] = 0) Then                   ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then   ; check precision lost
            set_precision()                 ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

VRNDSCALEPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF *src is a memory operand*
    THEN TMP_SRC ← BROADCAST32(SRC, VL, k1)
    ELSE TMP_SRC ← SRC
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← RoundToIntegerSP(TMP_SRC[i+31:i], imm8[7:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                                ; zeroing-masking
            DEST[i+31:i] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALEPS __m512 _mm512_roundscale_ps( __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_roundscale_round_ps( __m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_mask_roundscale_ps( __m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_mask_roundscale_round_ps( __m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_ps( __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_round_ps( __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 _mm256_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m256 _mm256_mask_roundscale_ps( __m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 _mm256_maskz_roundscale_ps( __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 _mm_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m128 _mm_mask_roundscale_ps( __m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 _mm_maskz_roundscale_ps( __mmask8 k, __m128 a, int imm);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2.

## VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEEX.NDS.LIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F	Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Rounds the single-precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The doubleword elements at bits 127:32 of the destination are copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the “Immediate Control Description” figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

$$\text{ROUND}(x) = 2^{-M} \cdot \text{Round\_to\_INT}(x \cdot 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x \cdot 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

EVEX encoded version: The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-25.

**Operation**

```

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC      ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]      ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]                          ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] ← round_to_nearest_even_integer(2M*SRC[31:0])
    01: TMP[31:0] ← round_to_equal_or_smaller_integer(2M*SRC[31:0])
    10: TMP[31:0] ← round_to_equal_or_larger_integer(2M*SRC[31:0])
    11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2M*SRC[31:0])
    ESAC;

    Dest[31:0] ← 2-M* TMP[31:0]             ; scale down back to 2-M
    if (imm8[3] = 0) Then                   ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then   ; check precision lost
            set_precision()                 ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

```

VRNDSCALESS (EVEX encoded version)

```

IF k1[0] or *no writemask*
    THEN  DEST[31:0] ← RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                             ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALESS __m128 _mm_roundscale_ss ( __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.

## VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4E /r VRSQRT14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4E /r VRSQRT14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4E /r VRSQRT14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ .

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

Operation

VRSQRT14PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[63:0]));
      ELSE DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+63:i]));
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Table 5-26. VRSQRT14PD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNaN_Indefinite	Including -INF
$X = -0$	-INF	
$X = +0$	+INF	
$X = +\text{INF}$	+0	

Intel C/C++ Compiler Intrinsic Equivalent

```
VRSQRT14PD __m512d __mm512_rsqrt14_pd( __m512d a);
VRSQRT14PD __m512d __mm512_mask_rsqrt14_pd(__m512d s, __mmask8 k, __m512d a);
VRSQRT14PD __m512d __mm512_maskz_rsqrt14_pd( __mmask8 k, __m512d a);
VRSQRT14PD __m256d __mm256_rsqrt14_pd( __m256d a);
VRSQRT14PD __m256d __mm256_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d __mm256_maskz_rsqrt14_pd( __mmask8 k, __m256d a);
VRSQRT14PD __m128d __mm128_rsqrt14_pd( __m128d a);
VRSQRT14PD __m128d __mm128_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d __mm128_maskz_rsqrt14_pd( __mmask8 k, __m128d a);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.



## VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 4F /r VRSQRT14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F	Computes the approximate reciprocal square root of the scalar double-precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the approximate reciprocal of the square roots of the scalar double-precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT14SD (EVEX version)

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  APPROXIMATE(1.0/ SQRT(SRC2[63:0]))

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  SRC1[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

Table 5-27. VRSQRT14SD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNaN_Indefinite	Including -INF
$X = -0$	-INF	
$X = +0$	+INF	
$X = +\text{INF}$	+0	

Intel C/C++ Compiler Intrinsic Equivalent

```
VRSQRT14SD __m128d __mm_rsqrt14_sd( __m128d a, __m128d b);
VRSQRT14SD __m128d __mm_mask_rsqrt14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRSQRT14SD __m128d __mm_maskz_rsqrt14_sd( __mmask8d m, __m128d a, __m128d b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.

## VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4E /r VRSQRT14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4E /r VRSQRT14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4E /r VRSQRT14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ .

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

**Operation****VRSQRT14PS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[31:0]));

ELSE DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+31:i]));

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

**Table 5-28. VRSQRT14PS Special Cases**

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNaN_Indefinite	Including -INF
$X = -0$	-INF	
$X = +0$	+INF	
$X = +\text{INF}$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT14PS \_\_m512 \_\_mm512\_rsqrt14\_ps( \_\_m512 a);

VRSQRT14PS \_\_m512 \_\_mm512\_mask\_rsqrt14\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VRSQRT14PS \_\_m512 \_\_mm512\_maskz\_rsqrt14\_ps( \_\_mmask16 k, \_\_m512 a);

VRSQRT14PS \_\_m256 \_\_mm256\_rsqrt14\_ps( \_\_m256 a);

VRSQRT14PS \_\_m256 \_\_mm256\_mask\_rsqrt14\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VRSQRT14PS \_\_m256 \_\_mm256\_maskz\_rsqrt14\_ps( \_\_mmask8 k, \_\_m256 a);

VRSQRT14PS \_\_m128 \_\_mm\_rsqrt14\_ps( \_\_m128 a);

VRSQRT14PS \_\_m128 \_\_mm\_mask\_rsqrt14\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VRSQRT14PS \_\_m128 \_\_mm\_maskz\_rsqrt14\_ps( \_\_mmask8 k, \_\_m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 4.

## VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 4F /r VRSQRT14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F	Computes the approximate reciprocal square root of the scalar single-precision floating-point value in xmm3/m32 and stores the result in the low doubleword element of xmm1 using writemask k1. Bits[127:32] of xmm2 is copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

Computes of the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $\infty$ , zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT14SS (EVEX version)

IF k1[0] or \*no writemask\*

THEN DEST[31:0]  $\leftarrow$  APPROXIMATE(1.0/ SQRT(SRC2[31:0]))

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0]  $\leftarrow$  0

FI;

FI;

DEST[127:32]  $\leftarrow$  SRC1[127:32]

DEST[MAXVL-1:128]  $\leftarrow$  0

Table 5-29. VRSQRT14SS Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNaN_Indefinite	Including -INF
$X = -0$	-INF	
$X = +0$	+INF	
$X = +\text{INF}$	+0	

Intel C/C++ Compiler Intrinsic Equivalent

```
VRSQRT14SS __m128 __mm_rsqrt14_ss( __m128 a, __m128 b);  
VRSQRT14SS __m128 __mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);  
VRSQRT14SS __m128 __mm_maskz_rsqrt14_ss( __mmask8 k, __m128 a, __m128 b);
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.

## VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 2C /r VSCALEFPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512F	Scale the packed double-precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1.
EVEX.NDS.256.66.0F38.W1 2C /r VSCALEFPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL AVX512F	Scale the packed double-precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1.
EVEX.NDS.512.66.0F38.W1 2C /r VSCALEFPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	A	V/V	AVX512F	Scale the packed double-precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a floating-point scale of the packed double-precision floating-point values in the first source operand by multiplying it by 2 power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

$$zmm1 := zmm2 * 2^{\text{floor}(zmm3)}$$

Floor(zmm3) means maximum integer value  $\leq$  zmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-30 and Table 5-31.

**Table 5-30. VSCALEFPD/SD/PS/SS Special Cases**

		Src2				Set IE
		$\pm$ NaN	+Inf	-Inf	0/Denorm/Norm	
Src1	$\pm$ QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNAN
	$\pm$ SNAN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
	$\pm$ Inf	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNAN or -INF
	$\pm$ 0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNAN or +INF
	Denorm/Norm	QNaN(Src2)	$\pm$ INF (Src1 sign)	$\pm$ 0 (Src1 sign)	Compute Result	IF Src2 is SNAN

**Table 5-31. Additional VSCALEFPD/SD Special Cases**

Special Case	Returned value	Faults
$ \text{result}  < 2^{-1074}$	$\pm 0$ or $\pm \text{Min-Denormal}$ (Src1 sign)	Underflow
$ \text{result}  \geq 2^{1024}$	$\pm \text{INF}$ (Src1 sign) or $\pm \text{Max-normal}$ (Src1 sign)	Overflow

**Operation**

SCALE(SRC1, SRC2)

```

{
  TMP_SRC2 ← SRC2
  TMP_SRC1 ← SRC1
  IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
  IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
  /* SRC2 is a 64 bits floating-point value */
  DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}

```

**VSCALEFPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 \*is register\*)

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[63:0]);

ELSE DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[i+63:i]);

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int);
VSCALEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m256d _mm256_scalef_round_pd(__m256d a, __m256d b, int);
VSCALEFPD __m256d _mm256_mask_scalef_round_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m256d _mm256_maskz_scalef_round_pd(__mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m128d _mm_scalef_round_pd(__m128d a, __m128d b, int);
VSCALEFPD __m128d _mm_mask_scalef_round_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFPD __m128d _mm_maskz_scalef_round_pd(__mmask8 k, __m128d a, __m128d b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).

Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E2.

## VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 2D /r VSCALEFSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	A	V/V	AVX512F	Scale the scalar double-precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a floating-point scale of the packed double-precision floating-point value in the first source operand by multiplying it by 2 power of the double-precision floating-point value in second source operand.

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{Floor}(\text{xmm3})}$$

Floor(xmm3) means maximum integer value  $\leq$  xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-30 and Table 5-31.

### Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 64 bits floating-point value */
    DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

**VSCALEFSD (EVEX encoded version)**

```

IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← SCALE(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                               ; zeroing-masking
                DEST[63:0] ← 0
        FI
    FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).  
Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E3.

## VSCALEFPS—Scale Packed Float32 Values With Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 2C /r VSCALEFPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512F	Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.
EVEX.NDS.256.66.0F38.W0 2C /r VSCALEFPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512F	Scale the packed single-precision values in ymm2 using floating point values from ymm3/m256/m32bcst. Under writemask k1.
EVEX.NDS.512.66.0F38.W0 2C /r VSCALEFPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	A	V/V	AVX512F	Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

$$zmm1 := zmm2 * 2^{\text{floor}(zmm3)}$$

Floor(zmm3) means maximum integer value  $\leq$  zmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Handling of special-case input values are listed in Table 5-30 and Table 5-32.

**Table 5-32. Additional VSCALEFPS/SS Special Cases**

Special Case	Returned value	Faults
$ \text{result}  < 2^{-149}$	$\pm 0$ or $\pm \text{Min-Denormal}$ (Src1 sign)	Underflow
$ \text{result}  \geq 2^{128}$	$\pm \text{INF}$ (Src1 sign) or $\pm \text{Max-normal}$ (Src1 sign)	Overflow

**Operation**

```

SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

```

**VSCALEFPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

```

IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[31:0]);
            ELSE DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[i+31:i]);
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFPS __m512 __mm512_scalef_round_ps(__m512 a, __m512 b, int);
VSCALEFPS __m512 __mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m512 __mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m256 __mm256_scalef_round_ps(__m256 a, __m256 b, int);
VSCALEFPS __m256 __mm256_mask_scalef_round_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m256 __mm256_maskz_scalef_round_ps(__mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m128 __mm_scalef_round_ps(__m128 a, __m128 b, int);
VSCALEFPS __m128 __mm_mask_scalef_round_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFPS __m128 __mm_maskz_scalef_round_ps(__mmask8 k, __m128 a, __m128 b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).  
Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E2.

## VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 2D /r VSCALEFSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	A	V/V	AVX512F	Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 power of the float32 value in second source operand.

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{floor}(\text{xmm3})}$$

Floor(xmm3) means maximum integer value  $\leq$  xmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-30 and Table 5-32.

**Operation**

```

SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

```

VSCALEFSS (EVEX encoded version)

```

IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] ← 0
        FI
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFSS __m128 __mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 __mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 __mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).  
Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E3.

## VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD vm32y {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z {k1}, ymm1	A	V/V	AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z {k1}, zmm1	A	V/V	AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

### Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address BASE\_ADDR and index vector VINDEXT, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.



- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  and alignment rules.  $N$  is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

#### VSCATTERDPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

    IF  $k1[j]$  OR \*no writemask\*

        THEN  $\text{MEM}[\text{BASE\_ADDR} + \text{SignExtend}(\text{VINDEX}[i+31:i]) * \text{SCALE} + \text{DISP}] \leftarrow$

$\text{SRC}[i+31:i]$

$k1[j] \leftarrow 0$

    FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$

#### VSCATTERDPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 64$

$k \leftarrow j * 32$

    IF  $k1[j]$  OR \*no writemask\*

        THEN  $\text{MEM}[\text{BASE\_ADDR} + \text{SignExtend}(\text{VINDEX}[k+31:k]) * \text{SCALE} + \text{DISP}] \leftarrow$

$\text{SRC}[i+63:i]$

$k1[j] \leftarrow 0$

    FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$

**VSCATTERQPS (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEK[k+63:k]) * SCALE + DISP] ←
            SRC[i+31:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**VSCATTERQPD (EVEX encoded versions)**

(KL, VL)= (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEK[i+63:i]) * SCALE + DISP] ←
            SRC[i+63:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCATTERDPD void __mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERDPD void __mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void __mm512_i32scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERDPS void __mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPD void __mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void __mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void __mm512_i64scatter_ps(void * base, __m512i vdx, __m256 a, int scale);
VSCATTERQPS void __mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m256 a, int scale);
VSCATTERDPD void __mm256_i32scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERDPD void __mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m256d a, int scale);
VSCATTERDPS void __mm256_i32scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERDPS void __mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void __mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void __mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void __mm256_i64scatter_ps(void * base, __m256i vdx, __m128 a, int scale);
VSCATTERQPS void __mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m128 a, int scale);
VSCATTERDPD void __mm_i32scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERDPD void __mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERDPS void __mm_i32scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERDPS void __mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERQPD void __mm_i64scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERQPD void __mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERQPS void __mm_i64scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERQPS void __mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);

```

**SIMD Floating-Point Exceptions**

Invalid, Overflow, Underflow, Precision, Denormal

**Other Exceptions**

See Exceptions Type E12.

## VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-bit Granularity

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F3A.W0 23 /r ib VSHUFF32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W0 23 /r ib VSHUFF32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F3A.W1 23 /r ib VSHUFF64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Shuffle 128-bit packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W1 23 /r ib VSHUFF64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Shuffle 128-bit packed double-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F3A.W0 43 /r ib VSHUFI32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W0 43 /r ib VSHUFI32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F3A.W1 43 /r ib VSHUFI64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W1 43 /r ib VSHUFI64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**256-bit Version:** Moves one of the two 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

**512-bit Version:** Moves two of the four 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.

**Operation**

```

Select2(SRC, control) {
CASE (control[0]) OF
  0:  TMP ← SRC[127:0];
  1:  TMP ← SRC[255:128];
ESAC;
RETURN TMP
}

```

```

Select4(SRC, control) {
CASE (control[1:0]) OF
  0:  TMP ← SRC[127:0];
  1:  TMP ← SRC[255:128];
  2:  TMP ← SRC[383:256];
  3:  TMP ← SRC[511:384];
ESAC;
RETURN TMP
}

```

**VSHUFF32x4 (EVEX versions)**

(KL, VL) = (8, 256), (16, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        THEN DEST[i+31:i] ← 0
      FI;
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VSHUFF64x2 (EVEX 512-bit version)**

(KL, VL) = (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN TMP\_SRC2[i+63:i] ← SRC2[63:0]

ELSE TMP\_SRC2[i+63:i] ← SRC2[i+63:i]

FI;

ENDFOR;

IF VL = 256

TMP\_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);

TMP\_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);

FI;

IF VL = 512

TMP\_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);

TMP\_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);

TMP\_DEST[383:256] ← Select4(TMP\_SRC2[511:0], imm8[5:4]);

TMP\_DEST[511:384] ← Select4(TMP\_SRC2[511:0], imm8[7:6]);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

THEN DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VSHUF132x4 (EVEX 512-bit version)**

(KL, VL) = (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN TMP\_SRC2[i+31:i] ← SRC2[31:0]

ELSE TMP\_SRC2[i+31:i] ← SRC2[i+31:i]

FI;

ENDFOR;

IF VL = 256

TMP\_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);

TMP\_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);

FI;

IF VL = 512

TMP\_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);

TMP\_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);

TMP\_DEST[383:256] ← Select4(TMP\_SRC2[511:0], imm8[5:4]);

TMP\_DEST[511:384] ← Select4(TMP\_SRC2[511:0], imm8[7:6]);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

```

    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            THEN DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

### VSHUFI64x2 (EVEX 512-bit version)

(KL, VL) = (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            THEN DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSHUFI32x4 __m512i_mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i_mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i_mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i_mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i_mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i_mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m512_mm512_shuffle_f32x4(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512_mm512_mask_shuffle_f32x4(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512_mm512_maskz_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFI64x2 __m512i_mm512_shuffle_i64x2(__m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i_mm512_mask_shuffle_i64x2(__m512i s, __mmask8 k, __m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i_mm512_maskz_shuffle_i64x2(__mmask8 k, __m512i a, __m512i b, int imm);
VSHUFF64x2 __m512d_mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d_mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d_mm512_maskz_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, int imm);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4NF.

#UD If EVEX.L'L = 0 for VSHUFF32x4/VSHUFF64x2.



## VTESTPD/VTESTPS—Packed Bit Test

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0E /r VTESTPS <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.
VEX.256.66.0F38.W0 0E /r VTESTPS <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.
VEX.128.66.0F38.W0 0F /r VTESTPD <i>xmm1, xmm2/m128</i>	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.
VEX.256.66.0F38.W0 0F /r VTESTPD <i>ymm1, ymm2/m256</i>	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

### Description

VTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M *reg* field.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

**Operation****VTESTPS (128-bit version)**

TEMP[127:0]  $\leftarrow$  SRC[127:0] AND DEST[127:0]  
 IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)  
   THEN ZF  $\leftarrow$  1;  
   ELSE ZF  $\leftarrow$  0;

TEMP[127:0]  $\leftarrow$  SRC[127:0] AND NOT DEST[127:0]  
 IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)  
   THEN CF  $\leftarrow$  1;  
   ELSE CF  $\leftarrow$  0;  
 DEST (unmodified)  
 AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;

**VTESTPS (VEX.256 encoded version)**

TEMP[255:0]  $\leftarrow$  SRC[255:0] AND DEST[255:0]  
 IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = TEMP[160] = TEMP[191] = TEMP[224] = TEMP[255] = 0)  
   THEN ZF  $\leftarrow$  1;  
   ELSE ZF  $\leftarrow$  0;

TEMP[255:0]  $\leftarrow$  SRC[255:0] AND NOT DEST[255:0]  
 IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = TEMP[160] = TEMP[191] = TEMP[224] = TEMP[255] = 0)  
   THEN CF  $\leftarrow$  1;  
   ELSE CF  $\leftarrow$  0;  
 DEST (unmodified)  
 AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;

**VTESTPD (128-bit version)**

TEMP[127:0]  $\leftarrow$  SRC[127:0] AND DEST[127:0]  
 IF (TEMP[63] = TEMP[127] = 0)  
   THEN ZF  $\leftarrow$  1;  
   ELSE ZF  $\leftarrow$  0;

TEMP[127:0]  $\leftarrow$  SRC[127:0] AND NOT DEST[127:0]  
 IF (TEMP[63] = TEMP[127] = 0)  
   THEN CF  $\leftarrow$  1;  
   ELSE CF  $\leftarrow$  0;  
 DEST (unmodified)  
 AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;

**VTESTPD (VEX.256 encoded version)**

TEMP[255:0]  $\leftarrow$  SRC[255:0] AND DEST[255:0]  
 IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)  
   THEN ZF  $\leftarrow$  1;  
   ELSE ZF  $\leftarrow$  0;

TEMP[255:0]  $\leftarrow$  SRC[255:0] AND NOT DEST[255:0]  
 IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)  
   THEN CF  $\leftarrow$  1;  
   ELSE CF  $\leftarrow$  0;  
 DEST (unmodified)  
 AF  $\leftarrow$  OF  $\leftarrow$  PF  $\leftarrow$  SF  $\leftarrow$  0;

## Intel C/C++ Compiler Intrinsic Equivalent

### VTESTPS

```
int _mm256_testz_ps (__m256 s1, __m256 s2);
int _mm256_testc_ps (__m256 s1, __m256 s2);
int _mm256_testnzc_ps (__m256 s1, __m128 s2);
int _mm_testz_ps (__m128 s1, __m128 s2);
int _mm_testc_ps (__m128 s1, __m128 s2);
int _mm_testnzc_ps (__m128 s1, __m128 s2);
```

### VTESTPD

```
int _mm256_testz_pd (__m256d s1, __m256d s2);
int _mm256_testc_pd (__m256d s1, __m256d s2);
int _mm256_testnzc_pd (__m256d s1, __m256d s2);
int _mm_testz_pd (__m128d s1, __m128d s2);
int _mm_testc_pd (__m128d s1, __m128d s2);
int _mm_testnzc_pd (__m128d s1, __m128d s2);
```

### Flags Affected

The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv ≠ 1111B.
	If VEX.W = 1 for VTESTPS or VTESTPD.

VZEROALL—Zero All YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.OF.WIG 77 VZEROALL	Z0	V/V	AVX	Zero all YMM registers.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

Description

The instruction zeros contents of all XMM or YMM registers.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

simd\_reg\_file[][] is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0 and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512. On a machine supporting Intel AVX but not Intel AVX-512, the width is "MAXVL".

VZEROALL (VEX.256 encoded version)

```
IF (64-bit mode)
    limit ← 15
ELSE
    limit ← 7
FOR i in 0 .. limit:
    simd_reg_file[i][MAXVL-1:0] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VZEROALL:      _mm256_zeroall()
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 8.

## VZERoupper—Zero Upper Bits of YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.0F.WIG 77 VZERoupper	Z0	V/V	AVX	Zero upper 128 bits of all YMM registers.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

The instruction zeros the bits in position 128 and higher of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

### Operation

`simd_reg_file[][]` is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0 and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512. On a machine supporting Intel AVX but not Intel AVX-512, the width is "MAXVL".

### VZERoupper

IF (64-bit mode)

limit  $\leftarrow$  15

ELSE

limit  $\leftarrow$  7

FOR i in 0 .. limit:

`simd_reg_file[i][MAXVL-1:128]  $\leftarrow$  0`

### Intel C/C++ Compiler Intrinsic Equivalent

VZERoupper: `_mm256_zeroupper()`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 8.

**WAIT/FWAIT—Wait**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9B	WAIT	Z0	Valid	Valid	Check pending unmasked floating-point exceptions.
9B	FWAIT	Z0	Valid	Valid	Check pending unmasked floating-point exceptions.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

CheckForPendingUnmaskedFloatingPointExceptions;

**FPU Flags Affected**

The C0, C1, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 09	WBINVD	Z0	Valid	Valid	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Writes back all modified cache lines in the processor's internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction. This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

### Operation

```
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.  
 #UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#UD                      If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)                      WBINVD cannot be executed at the virtual-8086 mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.



## WRFSBASE/WRGSBASE—Write FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Fea- ture Flag	Description
F3 OF AE /2 WRFSBASE <i>r32</i>	M	V/I	FSGSBASE	Load the FS base address with the 32-bit value in the source register.
F3 REX.W OF AE /2 WRFSBASE <i>r64</i>	M	V/I	FSGSBASE	Load the FS base address with the 64-bit value in the source register.
F3 OF AE /3 WRGSBASE <i>r32</i>	M	V/I	FSGSBASE	Load the GS base address with the 32-bit value in the source register.
F3 REX.W OF AE /3 WRGSBASE <i>r64</i>	M	V/I	FSGSBASE	Load the GS base address with the 64-bit value in the source register.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field.

The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

### Operation

FS/GS segment base address ← SRC;

### Flags Affected

None

### C/C++ Compiler Intrinsic Equivalent

```
WRFSBASE:    void _writefsbase_u32( unsigned int );
WRFSBASE:    _writefsbase_u64( unsigned __int64 );
WRGSBASE:    void _writegsbase_u32( unsigned int );
WRGSBASE:    _writegsbase_u64( unsigned __int64 );
```

### Protected Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

### Real-Address Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

#UD The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.FSGSBASE[bit 16] = 0. If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0
#GP(0)	If the source register contains a non-canonical address.

## WRMSR—Write to Model Specific Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 30	WRMSR	Z0	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 2, “Model-Specific Registers (MSRs)” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). Note that WRMSR to the IA32\_TSC\_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

### Operation

MSR[ECX] ← EDX:EAX;

### Flags Affected

None.

**Protected Mode Exceptions**

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the value in ECX specifies a reserved or unimplemented MSR address.</p> <p>If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.</p> <p>If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.</p>
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	<p>If the value in ECX specifies a reserved or unimplemented MSR address.</p> <p>If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.</p> <p>If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.</p>
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	The WRMSR instruction is not recognized in virtual-8086 mode.
--------	---

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## WRPKRU—Write Data to User Page Key Register

Opcode*	Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EF	WRPKRU	Z0	V/V	OSPKE	Writes EAX into PKRU.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Writes the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX and RAX are ignored.

### Operation

```
IF (ECX = 0 AND EDX = 0)
    THEN PKRU ← EAX;
    ELSE #GP(0);
FI;
```

### Flags Affected

None.

### C/C++ Compiler Intrinsic Equivalent

```
WRPKRU:    void _wrpkru(uint32_t);
```

### Protected Mode Exceptions

#GP(0)	If ECX ≠ 0. If EDX ≠ 0.
#UD	If the LOCK prefix is used. If CR4.PKE = 0.

### Real-Address Mode Exceptions

Same exceptions as in protected mode.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.

## XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE <sup>1</sup>	A hint used with an “XACQUIRE-enabled” instruction to start lock elision on the instruction memory operand address.
F3 XRELEASE	V/V	HLE	A hint used with an “XRELEASE-enabled” instruction to end lock elision on the instruction memory operand address.

### NOTES:

- Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

### Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The “MOV mem, reg” (Opcode 88H/89H) and “MOV mem, imm” (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.

## Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to

- REPNE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to

- REP/REPE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

## Operation

### XACQUIRE

IF XACQUIRE-enabled instruction

THEN

IF (HLE\_NEST\_COUNT < MAX\_HLE\_NEST\_COUNT) THEN

HLE\_NEST\_COUNT++

IF (HLE\_NEST\_COUNT = 1) THEN

HLE\_ACTIVE ← 1

IF 64-bit mode

THEN

restartRIP ← instruction pointer of the XACQUIRE-enabled instruction

ELSE

restartEIP ← instruction pointer of the XACQUIRE-enabled instruction

FI;

Enter HLE Execution (\* record register state, start tracking memory state \*)

FI; (\* HLE\_NEST\_COUNT = 1 \*)

IF ElisionBufferAvailable

THEN

Allocate elision buffer

Record address and data for forwarding and commit checking

Perform elision

ELSE

Perform lock acquire operation transactionally but without elision

FI;

ELSE (\* HLE\_NEST\_COUNT = MAX\_HLE\_NEST\_COUNT \*)

GOTO HLE\_ABORT\_PROCESSING

FI;

ELSE

Treat instruction as non-XACQUIRE F2H prefixed legacy instruction

FI;

**XRELEASE**

```

IF XRELEASE-enabled instruction
  THEN
    IF (HLE_NEST_COUNT > 0)
      THEN
        HLE_NEST_COUNT--
        IF lock address matches in elision buffer THEN
          IF lock satisfies address and value requirements THEN
            Deallocate elision buffer
          ELSE
            GOTO HLE_ABORT_PROCESSING
          FI;
        FI;
      IF (HLE_NEST_COUNT = 0)
        THEN
          IF NoAllocatedElisionBuffer
            THEN
              Try to commit transactional execution
              IF fail to commit transactional execution
                THEN
                  GOTO HLE_ABORT_PROCESSING;
                ELSE (* commit success *)
                  HLE_ACTIVE ← 0
                FI;
            ELSE
              GOTO HLE_ABORT_PROCESSING
            FI;
          FI;
        FI; (* HLE_NEST_COUNT > 0 *)
      ELSE
        Treat instruction as non-XRELEASE F3H prefixed legacy instruction
    FI;

```

(\* For any HLE abort condition encountered during HLE execution \*)

```

HLE_ABORT_PROCESSING:
  HLE_ACTIVE ← 0
  HLE_NEST_COUNT ← 0
  Restore architectural register state
  Discard memory updates performed in transaction
  Free any allocated lock elision buffers
  IF 64-bit mode
    THEN
      RIP ← restartRIP
    ELSE
      EIP ← restartEIP
  FI;
  Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END

```



## SIMD Floating-Point Exceptions

None

## Other Exceptions

#GP(0) If the use of prefix causes instruction length to exceed 15 bytes.

## XABORT — Transactional Abort

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
C6 F8 ib XABORT imm8	A	V/V	RTM	Causes an RTM abort if in RTM execution

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	imm8	NA	NA	NA

### Description

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

### Operation

#### XABORT

```
IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
FI;
```

(\* For any RTM abort condition encountered during RTM execution \*)

RTM\_ABORT\_PROCESSING:

```
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT ← 0;
    RTM_ACTIVE ← 0;
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP;
        ELSE
            EIP ← fallbackEIP;
    FI;
END
```

### Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

XABORT:     void \_xabort( unsigned int);

### SIMD Floating-Point Exceptions

None

**Other Exceptions**

#UD                    CUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.  
If LOCK prefix is used.

## XADD—Exchange and Add

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF C0 /r	XADD r/m8, r8	MR	Valid	Valid	Exchange r8 and r/m8; load sum into r/m8.
REX + OF C0 /r	XADD r/m8*, r8*	MR	Valid	N.E.	Exchange r8 and r/m8; load sum into r/m8.
OF C1 /r	XADD r/m16, r16	MR	Valid	Valid	Exchange r16 and r/m16; load sum into r/m16.
OF C1 /r	XADD r/m32, r32	MR	Valid	Valid	Exchange r32 and r/m32; load sum into r/m32.
REX.W + OF C1 /r	XADD r/m64, r64	MR	Valid	N.E.	Exchange r64 and r/m64; load sum into r/m64.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r, w)	NA	NA

### Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

### IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

### Operation

```
TEMP ← SRC + DEST;
SRC ← DEST;
DEST ← TEMP;
```

### Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## XBEGIN — Transactional Begin

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
C7 F8 XBEGIN rel16	A	V/V	RTM	Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.
C7 F8 XBEGIN rel32	A	V/V	RTM	Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	Offset	NA	NA	NA

### Description

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort.

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

### Operation

#### XBEGIN

```
IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
```

```
  THEN
```

```
    RTM_NEST_COUNT++
```

```
    IF RTM_NEST_COUNT = 1 THEN
```

```
      IF 64-bit Mode
```

```
        THEN
```

```
          fallbackRIP ← RIP + SignExtend64(IMM)
```

```
          (* RIP is instruction following XBEGIN instruction *)
```

```
        ELSE
```

```
          fallbackEIP ← EIP + SignExtend32(IMM)
```

```
          (* EIP is instruction following XBEGIN instruction *)
```

```
      FI;
```

```
    IF (64-bit mode)
```

```
      THEN IF (fallbackRIP is not canonical)
```

```
        THEN #GP(0)
```

```
      FI;
```

```
      ELSE IF (fallbackEIP outside code segment limit)
```

```
        THEN #GP(0)
```

```
      FI;
```

```
    FI;
```

```
    RTM_ACTIVE ← 1
```

```
    Enter RTM Execution (* record register state, start tracking memory state*)
```

```
  FI; (* RTM_NEST_COUNT = 1 *)
```

```

    ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
        GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
FI;
END

```

### Flags Affected

None

### Intel C/C++ Compiler Intrinsic Equivalent

XBEGIN:    unsigned int \_xbegin( void );

### SIMD Floating-Point Exceptions

None

### Protected Mode Exceptions

#UD                    CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.  
                       If LOCK prefix is used.

#GP(0)                If the fallback address is outside the CS segment.

### Real-Address Mode Exceptions

#GP(0)                If the fallback address is outside the address space 0000H and FFFFH.

#UD                    CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.  
                       If LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)                If the fallback address is outside the address space 0000H and FFFFH.

#UD                    CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.  
                       If LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-bit Mode Exceptions

- #UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.  
If LOCK prefix is used.
- #GP(0) If the fallback address is non-canonical.



## XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/ Leg Mode	Description
90+ <i>rw</i>	XCHG AX, <i>r16</i>	0	Valid	Valid	Exchange <i>r16</i> with AX.
90+ <i>rw</i>	XCHG <i>r16</i> , AX	0	Valid	Valid	Exchange AX with <i>r16</i> .
90+ <i>rd</i>	XCHG EAX, <i>r32</i>	0	Valid	Valid	Exchange <i>r32</i> with EAX.
REX.W + 90+ <i>rd</i>	XCHG RAX, <i>r64</i>	0	Valid	N.E.	Exchange <i>r64</i> with RAX.
90+ <i>rd</i>	XCHG <i>r32</i> , EAX	0	Valid	Valid	Exchange EAX with <i>r32</i> .
REX.W + 90+ <i>rd</i>	XCHG <i>r64</i> , RAX	0	Valid	N.E.	Exchange RAX with <i>r64</i> .
86 / <i>r</i>	XCHG <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Exchange <i>r8</i> (byte register) with byte from <i>r/m8</i> .
REX + 86 / <i>r</i>	XCHG <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	Exchange <i>r8</i> (byte register) with byte from <i>r/m8</i> .
86 / <i>r</i>	XCHG <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Exchange byte from <i>r/m8</i> with <i>r8</i> (byte register).
REX + 86 / <i>r</i>	XCHG <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Exchange byte from <i>r/m8</i> with <i>r8</i> (byte register).
87 / <i>r</i>	XCHG <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Exchange <i>r16</i> with word from <i>r/m16</i> .
87 / <i>r</i>	XCHG <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Exchange word from <i>r/m16</i> with <i>r16</i> .
87 / <i>r</i>	XCHG <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Exchange <i>r32</i> with doubleword from <i>r/m32</i> .
REX.W + 87 / <i>r</i>	XCHG <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Exchange <i>r64</i> with quadword from <i>r/m64</i> .
87 / <i>r</i>	XCHG <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Exchange doubleword from <i>r/m32</i> with <i>r32</i> .
REX.W + 87 / <i>r</i>	XCHG <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Exchange quadword from <i>r/m64</i> with <i>r64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
0	AX/EAX/RAX ( <i>r</i> , <i>w</i> )	opcode + <i>rd</i> ( <i>r</i> , <i>w</i> )	NA	NA
0	opcode + <i>rd</i> ( <i>r</i> , <i>w</i> )	AX/EAX/RAX ( <i>r</i> , <i>w</i> )	NA	NA
MR	ModRM: <i>r/m</i> ( <i>r</i> , <i>w</i> )	ModRM:reg ( <i>r</i> )	NA	NA
RM	ModRM:reg ( <i>w</i> )	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA

### Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**NOTE**

XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

**Operation**

TEMP  $\leftarrow$  DEST;  
 DEST  $\leftarrow$  SRC;  
 SRC  $\leftarrow$  TEMP;

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If either operand is in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## XEND — Transactional End

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D5 XEND	A	V/V	RTM	Specifies the end of an RTM code region.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

### Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

### Operation

#### XEND

```
IF (RTM_ACTIVE = 0) THEN
```

```
    SIGNAL #GP
```

```
ELSE
```

```
    RTM_NEST_COUNT--
```

```
    IF (RTM_NEST_COUNT = 0) THEN
```

```
        Try to commit transaction
```

```
        IF fail to commit transactional execution
```

```
            THEN
```

```
                GOTO RTM_ABORT_PROCESSING;
```

```
            ELSE (* commit success *)
```

```
                RTM_ACTIVE ← 0
```

```
        FI;
```

```
    FI;
```

```
FI;
```

```
(* For any RTM abort condition encountered during RTM execution *)
```

```
RTM_ABORT_PROCESSING:
```

```
    Restore architectural register state
```

```
    Discard memory updates performed in transaction
```

```
    Update EAX with status
```

```
    RTM_NEST_COUNT ← 0
```

```
    RTM_ACTIVE ← 0
```

```
    IF 64-bit Mode
```

```
        THEN
```

```
            RIP ← fallbackRIP
```

```
        ELSE
```

```
            EIP ← fallbackEIP
```

```
        FI;
```

```
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XEND:       void \_xend( void );

SIMD Floating-Point Exceptions

None

Other Exceptions

- #UD               CUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.  
                  If LOCK or 66H or F2H or F3H prefix is used.
- #GP(0)           If RTM\_ACTIVE = 0.

## XGETBV—Get Value of Extended Control Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP 0F 01 D0	XGETBV	Z0	Valid	Valid	Reads an XCR specified by ECX into EDX:EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, “Managing State Using the XSAVE Feature Set,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Use of any other value for ECX results in a general-protection (#GP) exception.

### Operation

EDX:EAX ← XCR[ECX];

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

XGETBV:     unsigned \_\_int64 \_xgetbv( unsigned int);

### Protected Mode Exceptions

- #GP(0)           If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD             If CPUID.01H:ECX.XSAVE[bit 26] = 0.  
If CR4.OSXSAVE[bit 18] = 0.  
If the LOCK prefix is used.

### Real-Address Mode Exceptions

- #GP(0)           If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD             If CPUID.01H:ECX.XSAVE[bit 26] = 0.  
If CR4.OSXSAVE[bit 18] = 0.  
If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### **Compatibility Mode Exceptions**

Same exceptions as in protected mode.

### **64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
D7	XLAT <i>m8</i>	Z0	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	Z0	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
REX.W + D7	XLATB	Z0	Valid	N.E.	Set AL to memory byte [RBX + unsigned AL].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operand” form and the “no-operand” form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a “short form” of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
IF AddressSize = 16
  THEN
    AL ← (DS:BX + ZeroExtend(AL));
  ELSE IF (AddressSize = 32)
    AL ← (DS:EBX + ZeroExtend(AL)); FI;
  ELSE (AddressSize = 64)
    AL ← (RBX + ZeroExtend(AL));
  FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
 If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#UD                      If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP                      If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
#SS                      If a memory operand effective address is outside the SS segment limit.  
#UD                      If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)                  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
#SS(0)                  If a memory operand effective address is outside the SS segment limit.  
#PF(fault-code)        If a page fault occurs.  
#UD                      If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)                  If a memory address referencing the SS segment is in a non-canonical form.  
#GP(0)                  If the memory address is in a non-canonical form.  
#PF(fault-code)        If a page fault occurs.  
#UD                      If the LOCK prefix is used.



## XOR—Logical Exclusive OR

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
34 <i>ib</i>	XOR AL, <i>imm8</i>	I	Valid	Valid	AL XOR <i>imm8</i> .
35 <i>iw</i>	XOR AX, <i>imm16</i>	I	Valid	Valid	AX XOR <i>imm16</i> .
35 <i>id</i>	XOR EAX, <i>imm32</i>	I	Valid	Valid	EAX XOR <i>imm32</i> .
REX.W + 35 <i>id</i>	XOR RAX, <i>imm32</i>	I	Valid	N.E.	RAX XOR <i>imm32</i> (sign-extended).
80 /6 <i>ib</i>	XOR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m8</i> XOR <i>imm8</i> .
REX + 80 /6 <i>ib</i>	XOR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m8</i> XOR <i>imm8</i> .
81 /6 <i>iw</i>	XOR <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	<i>r/m16</i> XOR <i>imm16</i> .
81 /6 <i>id</i>	XOR <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	<i>r/m32</i> XOR <i>imm32</i> .
REX.W + 81 /6 <i>id</i>	XOR <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	<i>r/m64</i> XOR <i>imm32</i> (sign-extended).
83 /6 <i>ib</i>	XOR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m16</i> XOR <i>imm8</i> (sign-extended).
83 /6 <i>ib</i>	XOR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m32</i> XOR <i>imm8</i> (sign-extended).
REX.W + 83 /6 <i>ib</i>	XOR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m64</i> XOR <i>imm8</i> (sign-extended).
30 / <i>r</i>	XOR <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	<i>r/m8</i> XOR <i>r8</i> .
REX + 30 / <i>r</i>	XOR <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	<i>r/m8</i> XOR <i>r8</i> .
31 / <i>r</i>	XOR <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	<i>r/m16</i> XOR <i>r16</i> .
31 / <i>r</i>	XOR <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	<i>r/m32</i> XOR <i>r32</i> .
REX.W + 31 / <i>r</i>	XOR <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	<i>r/m64</i> XOR <i>r64</i> .
32 / <i>r</i>	XOR <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	<i>r8</i> XOR <i>r/m8</i> .
REX + 32 / <i>r</i>	XOR <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	<i>r8</i> XOR <i>r/m8</i> .
33 / <i>r</i>	XOR <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	<i>r16</i> XOR <i>r/m16</i> .
33 / <i>r</i>	XOR <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	<i>r32</i> XOR <i>r/m32</i> .
REX.W + 33 / <i>r</i>	XOR <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	<i>r64</i> XOR <i>r/m64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

### Description

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← DEST XOR SRC;

### Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

### Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57/r XORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical XOR of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F.WIG 57 /r VXORPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 57 /r VXORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 57 /r VXORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 57 /r VXORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical XOR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation****VXORPD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0];

ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VXORPD (VEX.256 encoded version)**

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]

DEST[191:128] ← SRC1[191:128] BITWISE XOR SRC2[191:128]

DEST[255:192] ← SRC1[255:192] BITWISE XOR SRC2[255:192]

DEST[MAXVL-1:256] ← 0

**VXORPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]

DEST[MAXVL-1:128] ← 0

**XORPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] BITWISE XOR SRC[63:0]

DEST[127:64] ← DEST[127:64] BITWISE XOR SRC[127:64]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VXORPD \_\_m512d \_\_mm512\_xor\_pd (\_\_m512d a, \_\_m512d b);

VXORPD \_\_m512d \_\_mm512\_mask\_xor\_pd (\_\_m512d a, \_\_mmask8 m, \_\_m512d b);

VXORPD \_\_m512d \_\_mm512\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m512d a);

VXORPD \_\_m256d \_\_mm256\_xor\_pd (\_\_m256d a, \_\_m256d b);

VXORPD \_\_m256d \_\_mm256\_mask\_xor\_pd (\_\_m256d a, \_\_mmask8 m, \_\_m256d b);

VXORPD \_\_m256d \_\_mm256\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m256d a);

XORPD \_\_m128d \_\_mm\_xor\_pd (\_\_m128d a, \_\_m128d b);

VXORPD \_\_m128d \_\_mm\_mask\_xor\_pd (\_\_m128d a, \_\_mmask8 m, \_\_m128d b);

VXORPD \_\_m128d \_\_mm\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.

## XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 57 /r XORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 57 /r VXORPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 57 /r VXORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 57 /r VXORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 57 /r VXORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation****VXORPS (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b == 1) AND (SRC2 \*is memory\*)

THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0];

ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+31:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VXORPS (VEX.256 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]

DEST[159:128] ← SRC1[159:128] BITWISE XOR SRC2[159:128]

DEST[191:160] ← SRC1[191:160] BITWISE XOR SRC2[191:160]

DEST[223:192] ← SRC1[223:192] BITWISE XOR SRC2[223:192]

DEST[255:224] ← SRC1[255:224] BITWISE XOR SRC2[255:224].

DEST[MAXVL-1:256] ← 0

**VXORPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]

DEST[MAXVL-1:128] ← 0

**XORPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]

DEST[MAXVL-1:128] (Unmodified)

**Intel C/C++ Compiler Intrinsic Equivalent**

VXORPS \_\_m512 \_\_mm512\_xor\_ps (\_\_m512 a, \_\_m512 b);

VXORPS \_\_m512 \_\_mm512\_mask\_xor\_ps (\_\_m512 a, \_\_mmask16 m, \_\_m512 b);

VXORPS \_\_m512 \_\_mm512\_maskz\_xor\_ps (\_\_mmask16 m, \_\_m512 a);

VXORPS \_\_m256 \_\_mm256\_xor\_ps (\_\_m256 a, \_\_m256 b);

VXORPS \_\_m256 \_\_mm256\_mask\_xor\_ps (\_\_m256 a, \_\_mmask8 m, \_\_m256 b);

VXORPS \_\_m256 \_\_mm256\_maskz\_xor\_ps (\_\_mmask8 m, \_\_m256 a);

XORPS \_\_m128 \_\_mm\_xor\_ps (\_\_m128 a, \_\_m128 b);

VXORPS \_\_m128 \_\_mm\_mask\_xor\_ps (\_\_m128 a, \_\_mmask8 m, \_\_m128 b);

VXORPS \_\_m128 \_\_mm\_maskz\_xor\_ps (\_\_mmask8 m, \_\_m128 a);

#### **SIMD Floating-Point Exceptions**

None

#### **Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.



## XRSTOR—Restore Processor Extended States

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF AE /5	XRSTOR <i>mem</i>	M	Valid	Valid	Restore state components specified by EDX:EAX from <i>mem</i> .
NP REX.W + OF AE /5	XRSTOR64 <i>mem</i>	M	Valid	N.E.	Restore state components specified by EDX:EAX from <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.8, “Operation of XRSTOR,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP\_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If  $RFBM[i] = 0$ , XRSTOR does not update state component  $i$ .<sup>1</sup>
- If  $RFBM[i] = 1$  and bit  $i$  is clear in the XSTATE\_BV field in the XSAVE header, XRSTOR initializes state component  $i$ .
- If  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ , XRSTOR loads state component  $i$  from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR\_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component  $i$  for which  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ ; it tracks as modified any state component  $i$  for which  $RFBM[i] = 0$ .

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR\_INFO.

### Operation

$RFBM \leftarrow XCR0 \text{ AND } EDX:EAX$ ; /\* bitwise logical AND \*/

$COMP\_MASK \leftarrow XCOMP\_BV$  field from XSAVE header;

1. There is an exception if  $RFBM[1] = 0$  and  $RFBM[2] = 1$ . In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

RSTORMASK  $\leftarrow$  XSTATE\_BV field from XSAVE header;

IF COMPMASK[63] = 0

THEN

/\* Standard form of XRSTOR \*/

TO\_BE\_RESTORED  $\leftarrow$  RFBM AND RSTORMASK;

TO\_BE\_INITIALIZED  $\leftarrow$  RFBM AND NOT RSTORMASK;

IF TO\_BE\_RESTORED[0] = 1

THEN

load x87 state from legacy region of XSAVE area;

XINUSE[0]  $\leftarrow$  1;

ELSIF TO\_BE\_INITIALIZED[0] = 1

THEN

initialize x87 state;

XINUSE[0]  $\leftarrow$  0;

FI;

IF RFBM[1] = 1 OR RFBM[2] = 1

THEN load MXCSR from legacy region of XSAVE area;

FI;

IF TO\_BE\_RESTORED[1] = 1

THEN

load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR

XINUSE[1]  $\leftarrow$  1;

ELSIF TO\_BE\_INITIALIZED[1] = 1

THEN

set all XMM registers to 0; // this step does not initialize MXCSR

XINUSE[1]  $\leftarrow$  0;

FI;

FOR i  $\leftarrow$  2 TO 62

IF TO\_BE\_RESTORED[i] = 1

THEN

load XSAVE state component i at offset n from base of XSAVE area;

// n enumerated by CPUID(EAX=0DH,ECX=i):EBX)

XINUSE[i]  $\leftarrow$  1;

ELSIF TO\_BE\_INITIALIZED[i] = 1

THEN

initialize XSAVE state component i;

XINUSE[i]  $\leftarrow$  0;

FI;

ENDFOR;

ELSE

/\* Compacted form of XRSTOR \*/

IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0

THEN /\* compacted form not supported \*/

#GP(0);

FI;

FORMAT = COMPMASK AND 7FFFFFFF\_FFFFFFFFH;

RESTORE\_FEATURES = FORMAT AND RFBM;

```

TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT ← RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

```

```

IF TO_BE_RESTORED[0] = 1
    THEN

```

```

        load x87 state from legacy region of XSAVE area;
        XINUSE[0] ← 1;

```

```

    ELSIF TO_BE_INITIALIZED[0] = 1
        THEN

```

```

        initialize x87 state;
        XINUSE[0] ← 0;

```

```

FI;

```

```

IF TO_BE_RESTORED[1] = 1
    THEN

```

```

        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        XINUSE[1] ← 1;

```

```

    ELSIF TO_BE_INITIALIZED[1] = 1
        THEN

```

```

        set all XMM registers to 0;
        MXCSR ← 1F80H;
        XINUSE[1] ← 0;

```

```

FI;

```

```

NEXT_FEATURE_OFFSET = 576;           // Legacy area and XSAVE header consume 576 bytes

```

```

FOR i ← 2 TO 62

```

```

    IF FORMAT[i] = 1

```

```

        THEN

```

```

            IF TO_BE_RESTORED[i] = 1

```

```

                THEN

```

```

                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    XINUSE[i] ← 1;

```

```

                FI;

```

```

                NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);

```

```

            FI;

```

```

            IF TO_BE_INITIALIZED[i] = 1

```

```

                THEN

```

```

                    initialize XSAVE state component i;
                    XINUSE[i] ← 0;

```

```

                FI;

```

```

        ENDFOR;

```

```

FI;

```

```

XMODIFIED_BV ← NOT RFBM;

```

```

IF in VMX non-root operation

```

```

    THEN VMXNR ← 1;

```

```

    ELSE VMXNR ← 0;

```

```

FI;

```

```

LAXA ← linear address of XSAVE area;

```

```

XRSTOR_INFO ← <CPL,VMXNR,LAXA,COMPMASK>;

```

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

XRSTOR: `void _xrstor( void *, unsigned __int64);`

XRSTOR: `void _xrstor64( void *, unsigned __int64);`

## Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.</p> <p>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.</p> <p>If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## Real-Address Mode Exceptions

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.</p> <p>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.</p> <p>If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.</p>
-----	---

	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	<p>If a memory address is in a non-canonical form.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.</p> <p>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.</p> <p>If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## XRSTORS—Restore Processor Extended States Supervisor

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF C7 /3	XRSTORS <i>mem</i>	M	Valid	Valid	Restore state components specified by EDX:EAX from <i>mem</i> .
NP REX.W + OF C7 /3	XRSTORS64 <i>mem</i>	M	Valid	N.E.	Restore state components specified by EDX:EAX from <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32\_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.12, “Operation of XRSTORS,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, “Extended Region of an XSAVE Area”).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32\_XSS MSR.
- If  $RFBM[i] = 0$ , XRSTORS does not update state component  $i$ .
- If  $RFBM[i] = 1$  and bit  $i$  is clear in the XSTATE\_BV field in the XSAVE header, XRSTORS initializes state component  $i$ .
- If  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ , XRSTORS loads state component  $i$  from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR\_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component  $i$  for which  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ ; it tracks as modified any state component  $i$  for which  $RFBM[i] = 0$ .

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR\_INFO.

### Operation

```

RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;          /* bitwise logical OR and AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;

```

```

FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT ← RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        load x87 state from legacy region of XSAVE area;
        XINUSE[0] ← 1;
    ELSEIF TO_BE_INITIALIZED[0] = 1
        THEN
            initialize x87 state;
            XINUSE[0] ← 0;
    FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        XINUSE[1] ← 1;
    ELSEIF TO_BE_INITIALIZED[1] = 1
        THEN
            set all XMM registers to 0;
            MXCSR ← 1F80H;
            XINUSE[1] ← 0;
    FI;

NEXT_FEATURE_OFFSET = 576;           // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    XINUSE[i] ← 1;
                FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
        FI;
    IF TO_BE_INITIALIZED[i] = 1
        THEN
            initialize XSAVE state component i;
            XINUSE[i] ← 0;
        FI;
    ENDFOR;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← <CPL,VMXNR,LAXA,COMPMASK>;

```

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

XRSTORS: `void _xrstors( void *, unsigned __int64);`

XRSTORS64: `void _xrstors64( void *, unsigned __int64);`

## Protected Mode Exceptions

#GP(0)	<p>If CPL &gt; 0.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## Real-Address Mode Exceptions

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.



## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#GP(0)	<p>If <math>CPL &gt; 0</math>.</p> <p>If a memory address is in a non-canonical form.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If $CR0.TS[\text{bit } 3] = 1$ .
#UD	<p>If <math>CPUID.01H:ECX.XSAVE[\text{bit } 26] = 0</math> or <math>CPUID.(EAX=0DH,ECX=1):EAX.XSS[\text{bit } 3] = 0</math>.</p> <p>If <math>CR4.OSXSAVE[\text{bit } 18] = 0</math>.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## XSAVE—Save Processor Extended States

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF AE /4	XSAVE <i>mem</i>	M	Valid	Valid	Save state components specified by EDX:EAX to <i>mem</i> .
NP REX.W + OF AE /4	XSAVE64 <i>mem</i>	M	Valid	N.E.	Save state components specified by EDX:EAX to <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCRO.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.7, “Operation of XSAVE,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component *i* if and only if  $RFBM[i] = 1$ .<sup>1</sup>
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVE reads the XSTATE\_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If  $RFBM[i] = 1$ , XSAVE writes  $XSTATE\_BV[i]$  with the value of  $XINUSE[i]$ . (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) If  $RFBM[i] = 0$ , XSAVE writes  $XSTATE\_BV[i]$  with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE\_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

$RFBM \leftarrow XCRO \text{ AND } EDX:EAX$ ; /\* bitwise logical AND \*/

$OLD\_BV \leftarrow XSTATE\_BV$  field from XSAVE header;

IF  $RFBM[0] = 1$

THEN store x87 state into legacy region of XSAVE area;

FI;

IF  $RFBM[1] = 1$

THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR\_MASK

1. An exception is made for MXCSR and MXCSR\_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either  $RFBM[1]$  or  $RFBM[2]$  is 1.

```

FI;

IF RFBM[1] = 1 OR RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);

```

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVE:    void _xsave( void *, unsigned __int64);
XSAVE:    void _xsave64( void *, unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

### Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	<p>If the memory address is in a non-canonical form.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## XSAVEC—Save Processor Extended States with Compaction

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF C7 /4	XSAVEC <i>mem</i>	M	Valid	Valid	Save state components specified by EDX:EAX to <i>mem</i> with compaction.
NP REX.W + OF C7 /4	XSAVEC64 <i>mem</i>	M	Valid	N.E.	Save state components specified by EDX:EAX to <i>mem</i> with compaction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.10, “Operation of XSAVEC,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component *i* if and only if  $RFBM[i] = 1$  and  $XINUSE[i] = 1$ .<sup>1</sup> (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”)
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE\_BV field of the XSAVE header.<sup>2,3</sup> (See Section 13.4.2, “XSAVE Header.”) XSAVEC sets bit 63 of the XCOMP\_BV field and sets bits 62:0 of that field to  $RFBM[62:0]$ . XSAVEC does not write to any parts of the XSAVE header other than the XSTATE\_BV and XCOMP\_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

```

RFBM ← XCR0 AND EDX:EAX;           /* bitwise logical AND */
TO_BE_SAVED ← RFBM AND XINUSE;     /* bitwise logical AND */
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;

```

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as  $RFBM[1] = 1$ .
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE\_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE\_BV[1] to 1 as long as  $RFBM[1] = 1$ .

```

FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;           // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
        FI;
    ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← RFBM OR 80000000_00000000H;

```

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVEC:    void _xsavec( void *, unsigned __int64);
XSAVEC64:  void _xsavec64( void *, unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF AE /6 XSAVEOPT <i>mem</i>	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to <i>mem</i> , optimizing if possible.
NP REX.W + OF AE /6 XSAVEOPT64 <i>mem</i>	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to <i>mem</i> , optimizing if possible.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.9, “Operation of XSAVEOPT,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if  $RFBM[i] = 1$  and  $XINUSE[i] = 1$ .<sup>1</sup> (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVEOPT corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR\_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEOPT reads the XSTATE\_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If  $RFBM[i] = 1$ , XSAVEOPT writes  $XSTATE\_BV[i]$  with the value of  $XINUSE[i]$ . If  $RFBM[i] = 0$ , XSAVEOPT writes  $XSTATE\_BV[i]$  with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE\_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR\_INFO.

### Operation

$RFBM \leftarrow XCR0 \text{ AND } EDX:EAX; \quad /* \text{ bitwise logical AND } */$

1. There is an exception made for MXCSR and MXCSR\_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if  $RFBM[1] = 1$  or  $RFBM[2] = 1$ , regardless of the value of XINUSE.



```

OLD_BV ← XSTATE_BV field from XSAVE header;
TO_BE_SAVED ← RFBM AND XINUSE;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;

LAXA ← linear address of XSAVE area;
IF XRSTOR_INFO = <CPL,VMXNR,LAXA,00000000_00000000H>
    THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1]
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
    IF TO_BE_SAVED[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);

```

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVEOPT: void _xsaveopt( void *, unsigned __int64);
XSAVEOPT: void _xsaveopt64( void *, unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

## XSAVES—Save Processor Extended States Supervisor

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP OF C7 /5	XSAVES <i>mem</i>	M	Valid	Valid	Save state components specified by EDX:EAX to <i>mem</i> with compaction, optimizing if possible.
NP REX.W + OF C7 /5	XSAVES64 <i>mem</i>	M	Valid	N.E.	Save state components specified by EDX:EAX to <i>mem</i> with compaction, optimizing if possible.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32\_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.11, “Operation of XSAVES,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32\_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.<sup>1</sup> (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR\_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE\_BV field of the XSAVE header.<sup>2</sup> (See Section 13.4.2, “XSAVE Header.”) XSAVES sets bit 63 of the XCOMP\_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE\_BV and XCOMP\_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR\_INFO.

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE\_BV[1] to 1 as long as RFBM[1] = 1.

## Operation

```

RFBM ← (XCRO OR IA32_XSS) AND EDX:EAX;          /* bitwise logical OR and AND */
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← RFBM OR 80000000_00000000H;
TO_BE_SAVED ← RFBM AND XINUSE;
IF XRSTOR_INFO = (CPL,VMXNR,LAXA,COMPMASK)
    THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN
                    save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    IF i = 8          // state component 8 is for PT state
                        THEN IA32_RTIT_CTL.TraceEn[bit 0] ← 0;
                    FI;
                FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
        FI;
    ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← COMPMASK;

```

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVES:    void _xsaves( void *, unsigned __int64);
XSAVES64:  void _xsaves64( void *, unsigned __int64);

```

## Protected Mode Exceptions

#GP(0)	<p>If <math>CPL &gt; 0</math>.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If $CR0.TS[\text{bit } 3] = 1$ .
#UD	<p>If <math>CPUID.01H:ECX.XSAVE[\text{bit } 26] = 0</math> or <math>CPUID.(EAX=0DH,ECX=1):EAX.XSS[\text{bit } 3] = 0</math>.</p> <p>If <math>CR4.OSXSAVE[\text{bit } 18] = 0</math>.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## Real-Address Mode Exceptions

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p>
#NM	If $CR0.TS[\text{bit } 3] = 1$ .
#UD	<p>If <math>CPUID.01H:ECX.XSAVE[\text{bit } 26] = 0</math> or <math>CPUID.(EAX=0DH,ECX=1):EAX.XSS[\text{bit } 3] = 0</math>.</p> <p>If <math>CR4.OSXSAVE[\text{bit } 18] = 0</math>.</p> <p>If the LOCK prefix is used.</p>

## Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#GP(0)	<p>If <math>CPL &gt; 0</math>.</p> <p>If the memory address is in a non-canonical form.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If $CR0.TS[\text{bit } 3] = 1$ .
#UD	<p>If <math>CPUID.01H:ECX.XSAVE[\text{bit } 26] = 0</math> or <math>CPUID.(EAX=0DH,ECX=1):EAX.XSS[\text{bit } 3] = 0</math>.</p> <p>If <math>CR4.OSXSAVE[\text{bit } 18] = 0</math>.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an align-</p>

ment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## XSETBV—Set Extended Control Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP 0F 01 D1	XSETBV	Z0	Valid	Valid	Write the value in EDX:EAX to the XCR specified by ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, “Enabling the XSAVE Feature Set and XSAVE-Enabled Features,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

### Operation

$XCR[ECX] \leftarrow EDX:EAX;$

### Flags Affected

None.

### Intel C/C++ Compiler Intrinsic Equivalent

XSETBV: `void _xsetbv( unsigned int, unsigned __int64);`

### Protected Mode Exceptions

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If an invalid XCR is specified in ECX.</p> <p>If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.</p> <p>If an attempt is made to clear bit 0 of XCR0.</p> <p>If an attempt is made to set XCR0[2:1] to 10b.</p>
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>

### Real-Address Mode Exceptions

#GP	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0)	The XSETBV instruction is not recognized in virtual-8086 mode.
--------	--

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

Same exceptions as in protected mode.



## XTEST — Test If In Transactional Execution

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D6 XTEST	A	V/V	HLE or RTM	Test if executing in a transactional region

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

### Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

### Operation

#### XTEST

```
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF ← 0
    ELSE
        ZF ← 1
FI;
```

### Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

### Intel C/C++ Compiler Intrinsic Equivalent

XTEST: `int _xtest( void );`

### SIMD Floating-Point Exceptions

None

### Other Exceptions

#UD CPUID.(EAX=7, ECX=0):HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0.  
If LOCK prefix is used.

## GETSEC[CAPABILITIES] - Report the SMX Capabilities

Opcode	Instruction	Description
OF 37 (EAX = 0)	GETSEC[CAPABILITIES]	Report the SMX capabilities. The capabilities index is input in EBX with the result returned in EAX.

### Description

The GETSEC[CAPABILITIES] function returns a bit vector of supported GETSEC leaf functions. The CAPABILITIES leaf of GETSEC is selected with EAX set to 0 at entry. EBX is used as the selector for returning the bit vector field in EAX. GETSEC[CAPABILITIES] may be executed at all privilege levels, but the CR4.SMXE bit must be set or an undefined opcode exception (#UD) is returned.

With EBX = 0 upon execution of GETSEC[CAPABILITIES], EAX returns the a bit vector representing status on the presence of a Intel® TXT-capable chipset and the first 30 available GETSEC leaf functions. The format of the returned bit vector is provided in Table 6-3.

If bit 0 is set to 1, then an Intel® TXT-capable chipset has been sampled present by the processor. If bits in the range of 1-30 are set, then the corresponding GETSEC leaf function is available. If the bit value at a given bit index is 0, then the GETSEC leaf function corresponding to that index is unsupported and attempted execution results in a #UD.

Bit 31 of EAX indicates if further leaf indexes are supported. If the Extended Leafs bit 31 is set, then additional leaf functions are accessed by repeating GETSEC[CAPABILITIES] with EBX incremented by one. When the most significant bit of EAX is not set, then additional GETSEC leaf functions are not supported; indexing EBX to a higher value results in EAX returning zero.

**Table 6-3. Getsec Capability Result Encoding (EBX = 0)**

Field	Bit position	Description
Chipset Present	0	Intel® TXT-capable chipset is present.
Undefined	1	Reserved
ENTERACCS	2	GETSEC[ENTERACCS] is available.
EXITAC	3	GETSEC[EXITAC] is available.
SENDER	4	GETSEC[SENDER] is available.
SEXIT	5	GETSEC[SEXIT] is available.
PARAMETERS	6	GETSEC[PARAMETERS] is available.
SMCTRL	7	GETSEC[SMCTRL] is available.
WAKEUP	8	GETSEC[WAKEUP] is available.
Undefined	30:9	Reserved
Extended Leafs	31	Reserved for extended information reporting of GETSEC capabilities.

Operation

```
IF (CR4.SMXE=0)
    THEN #UD;
ELIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
IF (EBX=0) THEN
    BitVector← 0;
    IF (TXT chipset present)
        BitVector[Chipset present]← 1;
    IF (ENTERACCS Available)
        THEN BitVector[ENTERACCS]← 1;
    IF (EXITAC Available)
        THEN BitVector[EXITAC]← 1;
    IF (SENDER Available)
        THEN BitVector[SENDER]← 1;
    IF (SEXIT Available)
        THEN BitVector[SEXIT]← 1;
    IF (PARAMETERS Available)
        THEN BitVector[PARAMETERS]← 1;
    IF (SMCTRL Available)
        THEN BitVector[SMCTRL]← 1;
    IF (WAKEUP Available)
        THEN BitVector[WAKEUP]← 1;
    EAX← BitVector;
ELSE
    EAX← 0;
END;;
```

Flags Affected

None

Use of Prefixes

LOCK	Causes #UD
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ)
Operand size	Causes #UD
Segment overrides	Ignored
Address size	Ignored
REX	Ignored

Protected Mode Exceptions

#UD IF CR4.SMXE = 0.

Real-Address Mode Exceptions

#UD IF CR4.SMXE = 0.

Virtual-8086 Mode Exceptions

#UD IF CR4.SMXE = 0.

Compatibility Mode Exceptions

#UD IF CR4.SMXE = 0.

## 64-Bit Mode Exceptions

#UD IF CR4.SMXE = 0.

## VM-exit Condition

Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[ENTERACCS] - Execute Authenticated Chipset Code

Opcode	Instruction	Description
0F 37 (EAX = 2)	GETSEC[ENTERACCS]	Enter authenticated code execution mode. EBX holds the authenticated code module physical base address. ECX holds the authenticated code module size (bytes).

### Description

The GETSEC[ENTERACCS] function loads, authenticates and executes an authenticated code module using an Intel® TXT platform chipset's public key. The ENTERACCS leaf of GETSEC is selected with EAX set to 2 at entry.

There are certain restrictions enforced by the processor for the execution of the GETSEC[ENTERACCS] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled, that is, CR0.CD and CR0.NW bits must be 0.
- For processor packages containing more than one logical processor, CR0.CD is checked to ensure consistency between enabled logical processors.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not already be in authenticated code execution mode as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENDER] instruction without a subsequent exiting using GETSEC[EXITAC]).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To insure consistent handling of SIPI messages, the processor executing the GETSEC[ENTERACCS] instruction must also be designated the BSP (boot-strap processor) as defined by IA32\_APIC\_BASE.BSP (Bit 8).

Failure to conform to the above conditions results in the processor signaling a general protection exception.

Prior to execution of the ENTERACCS leaf, other logical processors, i.e., RLPs, in the platform must be:

- Idle in a wait-for-SIPI state (as initiated by an INIT assertion or through reset for non-BSP designated processors), or
- In the SENTER sleep state as initiated by a GETSEC[SENDER] from the initiating logical processor (ILP).

If other logical processor(s) in the same package are not idle in one of these states, execution of ENTERACCS signals a general protection exception. The same requirement and action applies if the other logical processor(s) of the same package do not have CR0.CD = 0.

A successful execution of ENTERACCS results in the ILP entering an authenticated code execution mode. Prior to reaching this point, the processor performs several checks. These include:

- Establish and check the location and size of the specified authenticated code module to be executed by the processor.
- Inhibit the ILP's response to the external events: INIT, A20M, NMI and SMI.
- Broadcast a message to enable protection of memory and I/O from other processor agents.
- Load the designated code module into an authenticated code execution area.
- Isolate the contents of the authenticated code execution area from further state modification by external agents.
- Authenticate the authenticated code module.
- Initialize the initiating logical processor state based on information contained in the authenticated code module header.
- Unlock the Intel® TXT-capable chipset private configuration space and TPM locality 3 space.

- Begin execution in the authenticated code module at the defined entry point.

The GETSEC[ENTERACCS] function requires two additional input parameters in the general purpose registers EBX and ECX. EBX holds the authenticated code (AC) module physical base address (the AC module must reside below 4 GBytes in physical address space) and ECX holds the AC module size (in bytes). The physical base address and size are used to retrieve the code module from system memory and load it into the internal authenticated code execution area. The base physical address is checked to verify it is on a modulo-4096 byte boundary. The size is verified to be a multiple of 64, that it does not exceed the internal authenticated code execution area capacity (as reported by GETSEC[CAPABILITIES]), and that the top address of the AC module does not exceed 32 bits. An error condition results in an abort of the authenticated code execution launch and the signaling of a general protection exception.

As an integrity check for proper processor hardware operation, execution of GETSEC[ENTERACCS] will also check the contents of all the machine check status registers (as reported by the MSRs IA32\_MCI\_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32\_MCG\_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must not be asserted, indicating that no machine check exception processing is currently in progress. These checks are performed prior to initiating the load of the authenticated code module. Any outstanding valid uncorrectable machine check error condition present in these status registers at this point will result in the processor signaling a general protection violation.

The ILP masks the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. This masking remains active until optionally unmasked by GETSEC[EXITAC] (this defined unmasking behavior assumes GETSEC[ENTERACCS] was not executed by a prior GETSEC[SENDER]). The purpose of this masking control is to prevent exposure to existing external event handlers that may not be under the control of the authenticated code module.

The ILP sets an internal flag to indicate it has entered authenticated code execution mode. The state of the A20M pin is likewise masked and forced internally to a de-asserted state so that any external assertion is not recognized during authenticated code execution mode.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) access and I/O originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Prior to launching the authenticated execution module using GETSEC[ENTERACCS] or GETSEC[SENDER], the processor's MTRRs (Memory Type Range Registers) must first be initialized to map out the authenticated RAM addresses as WB (writeback). Failure to do so may affect the ability for the processor to maintain isolation of the loaded authenticated code module. If the processor detected this requirement is not met, it will signal an Intel® TXT reset condition with an error code during the loading of the authenticated code module.

While physical addresses within the load module must be mapped as WB, the memory type for locations outside of the module boundaries must be mapped to one of the supported memory types as returned by GETSEC[PARAMETERS] (or UC as default).

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

At the successful completion of GETSEC[ENTERACCS], the architectural state of the processor is partially initialized from contents held in the header of the authenticated code module. The processor GDTR, CS, and DS selectors are initialized from fields within the authenticated code module. Since the authenticated code module must be relocatable, all address references must be relative to the authenticated code module base address in EBX. The processor GDTR base value is initialized to the AC module header field GDTBasePtr + module base address held in EBX and the GDTR limit is set to the value in the GDTLimit field. The CS selector is initialized to the AC module header SegSel field, while the DS selector is initialized to CS + 8. The segment descriptor fields are implicitly initialized to BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, S=1, read/write access for DS, and execute/read access for CS. The processor begins the authenticated code module execution with the EIP set to the AC module header EntryPoint field + module base address (EBX). The AC module based fields used for initializing the processor state are checked for consistency and any failure results in a shutdown condition.

A summary of the register state initialization after successful completion of GETSEC[ENTERACCS] is given for the processor in Table 6-4. The paging is disabled upon entry into authenticated code execution mode. The authenticated code module is loaded and initially executed using physical addresses. It is up to the system software after execution of GETSEC[ENTERACCS] to establish a new (or restore its previous) paging environment with an appropriate mapping to meet new protection requirements. EBP is initialized to the authenticated code module base physical address for initial execution in the authenticated environment. As a result, the authenticated code can reference EBP for relative address based references, given that the authenticated code module must be position independent.

**Table 6-4. Register State Initialization after GETSEC[ENTERACCS]**

Register State	Initialization Status	Comment
CR0	PG←0, AM←0, WP←0: Others unchanged	Paging, Alignment Check, Write-protection are disabled.
CR4	MCE←0: Others unchanged	Machine Check Exceptions disabled.
EFLAGS	00000002H	
IA32_EFER	0H	IA-32e mode disabled.
EIP	AC.base + EntryPoint	AC.base is in EBX as input to GETSEC[ENTERACCS].
[E R]BX	Pre-ENTERACCS state: Next [E R]IP prior to GETSEC[ENTERACCS]	Carry forward 64-bit processor state across GETSEC[ENTERACCS].
ECX	Pre-ENTERACCS state: [31:16]=GDTR.limit; [15:0]=CS.sel	Carry forward processor state across GETSEC[ENTERACCS].
[E R]DX	Pre-ENTERACCS state: GDTR base	Carry forward 64-bit processor state across GETSEC[ENTERACCS].
EBP	AC.base	
CS	Sel=[SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	
DS	Sel=[SegSel] +8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	
GDTR	Base= AC.base (EBX) + [GDTBasePtr], Limit=[GDTLimit]	
DR7	00000400H	
IA32_DEBUGCTL	0H	
IA32_MISC_ENABLE	See Table 6-5 for example.	The number of initialized fields may change due to processor implementation.

The segmentation related processor state that has not been initialized by GETSEC[ENTERACCS] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in ES, SS, FS, GS, TR, and LDTR might not be valid.

The MSR IA32\_EFER is also unconditionally cleared as part of the processor state initialized by ENTERACCS. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be reestablished in order to establish IA-32e mode while operating in authenticated code execution mode.

Debug exception and trap related signaling is also disabled as part of GETSEC[ENTERACCS]. This is achieved by resetting DR7, TF in EFLAGS, and the MSR IA32\_DEBUGCTL. These debug functions are free to be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly initialized following

entry into authenticated code execution mode. Also, any pending single-step trap condition will have been cleared upon entry into this mode.

The IA32\_MISC\_ENABLE MSR is initialized upon entry into authenticated execution mode. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings (See the footnote for Table 6-5.). The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. One of the impacts of initializing this MSR is any previous condition established by the MONITOR instruction will be cleared.

To support the possible return to the processor architectural state prior to execution of GETSEC[ENTERACCS], certain critical processor state is captured and stored in the general-purpose registers at instruction completion. [E|R]BX holds effective address ([E|R]IP) of the instruction that would execute next after GETSEC[ENTERACCS], ECX[15:0] holds the CS selector value, ECX[31:16] holds the GDTR limit field, and [E|R]DX holds the GDTR base field. The subsequent authenticated code can preserve the contents of these registers so that this state can be manually restored if needed, prior to exiting authenticated code execution mode with GETSEC[EXITAC]. For the processor state after exiting authenticated code execution mode, see the description of GETSEC[SEXIT].

**Table 6-5. IA32\_MISC\_ENABLE MSR Initialization<sup>1</sup> by ENTERACCS and SENTER**

Field	Bit position	Description
Fast strings enable	0	Clear to 0.
FOPCODE compatibility mode enable	2	Clear to 0.
Thermal monitor enable	3	Set to 1 if other thermal monitor capability is not enabled. <sup>2</sup>
Split-lock disable	4	Clear to 0.
Bus lock on cache line splits disable	8	Clear to 0.
Hardware prefetch disable	9	Clear to 0.
GV1/2 legacy enable	15	Clear to 0.
MONITOR/MWAIT s/m enable	18	Clear to 0.
Adjacent sector prefetch disable	19	Clear to 0.

#### NOTES:

1. The number of IA32\_MISC\_ENABLE fields that are initialized may vary due to processor implementations.
2. ENTERACCS (and SENTER) initialize the state of processor thermal throttling such that at least a minimum level is enabled. If thermal throttling is already enabled when executing one of these GETSEC leaves, then no change in the thermal throttling control settings will occur. If thermal throttling is disabled, then it will be enabled via setting of the thermal throttle control bit 3 as a result of executing these GETSEC leaves.

The IDTR will also require reloading with a new IDT context after entering authenticated code execution mode, before any exceptions or the external interrupts INTR and NMI can be handled. Since external interrupts are re-enabled at the completion of authenticated code execution mode (as terminated with EXITAC), it is recommended that a new IDT context be established before this point. Until such a new IDT context is established, the programmer must take care in not executing an INT n instruction or any other operation that would result in an exception or trap signaling.

Prior to completion of the GETSEC[ENTERACCS] instruction and after successful authentication of the AC module, the private configuration space of the Intel TXT chipset is unlocked. The authenticated code module alone can gain access to this normally restricted chipset state for the purpose of securing the platform.

Once the authenticated code module is launched at the completion of GETSEC[ENTERACCS], it is free to enable interrupts by setting EFLAGS.IF and enable NMI by execution of IRET. This presumes that it has re-established interrupt handling support through initialization of the IDT, GDT, and corresponding interrupt handling code.



**Operation in a Uni-Processor Platform**

(\* The state of the internal flag ACMODEFLAG persists across instruction boundary \*)

```

IF (CR4.SMXE=0)
    THEN #UD;
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or
    (CRO.PE=0) or (CRO.CD=1) or (CRO.NW=1) or (CRO.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or
    (TXT chipset not present) or
    (ACMODEFLAG=1) or (IN_SMM=1))
    THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF (IA32_MCG[I].STATUS = uncorrectable error)
            THEN #GP(0);
    OD;
FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);
ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) ≠ 0) or ((ACSIZE MOD 64) ≠ 0) or (ACSIZE < minimum module size) OR (ACSIZE > authenticated RAM
capacity)) or ((ACBASE+ACSIZE) > (232-1)))
    THEN #GP(0);
IF (secondary thread(s) CRO.CD = 1) or ((secondary thread(s) NOT(wait-for-SIPI)) and
    (secondary thread(s) not in SENTER sleep state)
    THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
IA32_MISC_ENABLE← (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M← 0;
IA32_DEBUGCTL← 0;
Invalidate processor TLB(s);
Drain Outgoing Transactions;
ACMODEFLAG← 1;
SignalTXTMessage(ProcessorHold);
Load the internal ACRAM based on the AC module size;
(* Ensure that all ACRAM loads hit Write Back memory space *)
IF (ACRAM memory type ≠ WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version isnot supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
(* Authenticate the AC Module and shutdown with an error if it fails *)
KEY← GETKEY(ACRAM, ACRAM);
KEYHASH← HASH(KEY);
CSKEYHASH← READ(TXT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE← DECRYPT(ACRAM, ACRAM, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)

```

```

FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I]← SIGNATURE[I];
COMPUTEDSIGNATURE← HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I]← COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL← ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN ACEntryPoint← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint← ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) OR (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size))) THEN TXT-SHUTDOWN(#BadACMFormat);
IF (ACRAM[GDTLimit] & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) OR (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) OR (ACRAM[SegSel].RPL≠0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
CRO.[PG.AM.WP]← 0;
CR4.MCE← 0;
EFLAGS← 00000002h;
IA32_EFER← 0h;
[EIP]BX← [EIP]IP of the instruction after GETSEC[ENTERACCS];
ECX← Pre-GETSEC[ENTERACCS] GDT.limit:CS.sel;
[EIP]DX← Pre-GETSEC[ENTERACCS] GDT.base;
EBP← ACBASE;
GDTR.BASE← ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT← ACRAM[GDTLimit];
CS.SEL← ACRAM[SegSel];
CS.BASE← 0;
CS.LIMIT← FFFFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← ACRAM[SegSel]+8;
DS.BASE← 0;
DS.LIMIT← FFFFFFFh;
DS.G← 1;
DS.D← 1;
DS.AR← 93h;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP← ACEntryPoint;
END;

```

## Flags Affected

All flags are cleared.

## Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

## Protected Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	<p>If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL &gt; 0 or EFLAGS.VM = 1.</p> <p>If a Intel® TXT-capable chipset is not present.</p> <p>If in VMX root operation.</p> <p>If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32_APIC_BASE.BSP.</p> <p>If the processor is already in authenticated code execution mode.</p> <p>If the processor is in SMM.</p> <p>If a valid uncorrectable machine check error is logged in IA32_MC[I]_STATUS.</p> <p>If the authenticated code base is not on a 4096 byte boundary.</p> <p>If the authenticated code size &gt; processor internal authenticated code area capacity.</p> <p>If the authenticated code size is not modulo 64.</p> <p>If other enabled logical processor(s) of the same package CR0.CD = 1.</p> <p>If other enabled logical processor(s) of the same package are not in the wait-for-SIPI or SENTER sleep state.</p>

## Real-Address Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[ENTERACCS] is not recognized in real-address mode.

## Virtual-8086 Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[ENTERACCS] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[ENTERACCS] is not recognized in virtual-8086 mode.

## Compatibility Mode Exceptions

All protected mode exceptions apply.

#GP	IF AC code module does not reside in physical address below $2^{32} - 1$ .
-----	--

## 64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP	IF AC code module does not reside in physical address below $2^{32} - 1$ .
-----	--

**VM-exit Condition**

Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[EXITAC]—Exit Authenticated Code Execution Mode

Opcode	Instruction	Description
0F 37 (EAX=3)	GETSEC[EXITAC]	Exit authenticated code execution mode. RBX holds the Near Absolute Indirect jump target and EDX hold the exit parameter flags.

### Description

The GETSEC[EXITAC] leaf function exits the ILP out of authenticated code execution mode established by GETSEC[ENTERACCS] or GETSEC[SENDER]. The EXITAC leaf of GETSEC is selected with EAX set to 3 at entry. EBX (or RBX, if in 64-bit mode) holds the near jump target offset for where the processor execution resumes upon exiting authenticated code execution mode. EDX contains additional parameter control information. Currently only an input value of 0 in EDX is supported. All other EDX settings are considered reserved and result in a general protection violation.

GETSEC[EXITAC] can only be executed if the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0. The processor must also be in authenticated code execution mode. To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it is in SMM or in VMX operation. A violation of these conditions results in a general protection violation.

Upon completion of the GETSEC[EXITAC] operation, the processor unmask responses to external event signals INIT#, NMI#, and SMI#. This unmasking is performed conditionally, based on whether the authenticated code execution mode was entered via execution of GETSEC[SENDER] or GETSEC[ENTERACCS]. If the processor is in authenticated code execution mode due to the execution of GETSEC[SENDER], then these external event signals will remain masked. In this case, A20M is kept disabled in the measured environment until the measured environment executes GETSEC[SEXIT]. INIT# is unconditionally unmasked by EXITAC. Note that any events that are pending, but have been blocked while in authenticated code execution mode, will be recognized at the completion of the GETSEC[EXITAC] instruction if the pin event is unmasked.

The intent of providing the ability to optionally leave the pin events SMI#, and NMI# masked is to support the completion of a measured environment bring-up that makes use of VMX. In this envisioned security usage scenario, these events will remain masked until an appropriate virtual machine has been established in order to field servicing of these events in a safer manner. Details on when and how events are masked and unmasked in VMX operation are described in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*. It should be cautioned that if no VMX environment is to be activated following GETSEC[EXITAC], that these events will remain masked until the measured environment is exited with GETSEC[SEXIT]. If this is not desired then the GETSEC function SMCTRL(0) can be used for unmasking SMI# in this context. NMI# can be correspondingly unmasked by execution of IRET.

A successful exit of the authenticated code execution mode requires the ILP to perform additional steps as outlined below:

- Invalidate the contents of the internal authenticated code execution area.
- Invalidate processor TLBs.
- Clear the internal processor AC Mode indicator flag.
- Re-lock the TPM locality 3 space.
- Unlock the Intel® TXT-capable chipset memory and I/O protections to allow memory and I/O activity by other processor agents.
- Perform a near absolute indirect jump to the designated instruction location.

The content of the authenticated code execution area is invalidated by hardware in order to protect it from further use or visibility. This internal processor storage area can no longer be used or relied upon after GETSEC[EXITAC]. Data structures need to be re-established outside of the authenticated code execution area if they are to be referenced after EXITAC. Since addressed memory content formerly mapped to the authenticated code execution area may no longer be coherent with external system memory after EXITAC, processor TLBs in support of linear to physical address translation are also invalidated.

Upon completion of GETSEC[EXITAC] a near absolute indirect transfer is performed with EIP loaded with the contents of EBX (based on the current operating mode size). In 64-bit mode, all 64 bits of RBX are loaded into RIP

if REX.W precedes GETSEC[EXITAC]. Otherwise RBX is treated as 32 bits even while in 64-bit mode. Conventional CS limit checking is performed as part of this control transfer. Any exception conditions generated as part of this control transfer will be directed to the existing IDT; thus it is recommended that an IDTR should also be established prior to execution of the EXITAC function if there is a need for fault handling. In addition, any segmentation related (and paging) data structures to be used after EXITAC should be re-established or validated by the authenticated code prior to EXITAC.

In addition, any segmentation related (and paging) data structures to be used after EXITAC need to be re-established and mapped outside of the authenticated RAM designated area by the authenticated code prior to EXITAC. Any data structure held within the authenticated RAM allocated area will no longer be accessible after completion by EXITAC.

## Operation

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

```

IF (CR4.SMXE=0)
    THEN #UD;
ELSIF ( in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or ( in 64-bit mode) and ( RBX is non-canonical) )
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (ACMODEFLAG=0) or (IN_SMM=1)) or (EDX ≠ 0))
    THEN #GP(0);
IF (OperandSize = 32)
    THEN tempEIP← EBX;
ELSIF (OperandSize = 64)
    THEN tempEIP← RBX;
ELSE
    tempEIP← EBX AND 0000FFFFH;
IF (tempEIP > code segment limit)
    THEN #GP(0);
Invalidate ACRAM contents;
Invalidate processor TLB(s);
Drain outgoing messages;
SignalTXTMsg(CloseLocality3);
SignalTXTMsg(LockSMRAM);
SignalTXTMsg(ProcessorRelease);
Unmask INIT;
IF (SENTERFLAG=0)
    THEN Unmask SMI, INIT, NMI, and A20M pin event;
ELSEIF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ACMODEFLAG← 0;
EIP← tempEIP;
END;
```

## Flags Affected

None.

## Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.

Segment overrides Ignored.  
 Address size Ignored.  
 REX.W Sets 64-bit mode Operand size attribute.

### Protected Mode Exceptions

#UD If CR4.SMXE = 0.  
 If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].  
 #GP(0) If CR0.PE = 0 or CPL>0 or EFLAGS.VM = 1.  
 If in VMX root operation.  
 If the processor is not currently in authenticated code execution mode.  
 If the processor is in SMM.  
 If any reserved bit position is set in the EDX parameter register.

### Real-Address Mode Exceptions

#UD If CR4.SMXE = 0.  
 If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].  
 #GP(0) GETSEC[EXITAC] is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD If CR4.SMXE = 0.  
 If GETSEC[EXITAC] is not reported as supported by GETSEC[CAPABILITIES].  
 #GP(0) GETSEC[EXITAC] is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

All protected mode exceptions apply.

### 64-Bit Mode Exceptions

All protected mode exceptions apply.  
 #GP(0) If the target address in RBX is not in a canonical form.

### VM-Exit Condition

Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[SENTER]—Enter a Measured Environment

Opcode	Instruction	Description
0F 37 (EAX=4)	GETSEC[SENTER]	Launch a measured environment. EBX holds the SINIT authenticated code module physical base address. ECX holds the SINIT authenticated code module size (bytes). EDX controls the level of functionality supported by the measured environment launch.

### Description

The GETSEC[SENTER] instruction initiates the launch of a measured environment and places the initiating logical processor (ILP) into the authenticated code execution mode. The SENTER leaf of GETSEC is selected with EAX set to 4 at execution. The physical base address of the AC module to be loaded and authenticated is specified in EBX. The size of the module in bytes is specified in ECX. EDX controls the level of functionality supported by the measured environment launch. To enable the full functionality of the protected environment launch, EDX must be initialized to zero.

The authenticated code base address and size parameters (in bytes) are passed to the GETSEC[SENTER] instruction using EBX and ECX respectively. The ILP evaluates the contents of these registers according to the rules for the AC module address in GETSEC[ENTERACCS]. AC module execution follows the same rules, as set by GETSEC[ENTERACCS].

The launching software must ensure that the TPM.ACCESS\_0.activeLocality bit is clear before executing the GETSEC[SENTER] instruction.

There are restrictions enforced by the processor for execution of the GETSEC[SENTER] instruction:

- Execution is not allowed unless the processor is in protected mode or IA-32e mode with CPL = 0 and EFLAGS.VM = 0.
- Processor cache must be available and not disabled using the CR0.CD and NW bits.
- For enforcing consistency of operation with numeric exception reporting using Interrupt 16, CR0.NE must be set.
- An Intel TXT-capable chipset must be present as communicated to the processor by sampling of the power-on configuration capability field after reset.
- The processor can not be in authenticated code execution mode or already in a measured environment (as launched by a previous GETSEC[ENTERACCS] or GETSEC[SENTER] instruction).
- To avoid potential operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or VMX operation.
- To insure consistent handling of SIPI messages, the processor executing the GETSEC[SENTER] instruction must also be designated the BSP (boot-strap processor) as defined by A32\_APIC\_BASE.BSP (Bit 8).
- EDX must be initialized to a setting supportable by the processor. Unless enumeration by the GETSEC[PARAM-ETERS] leaf reports otherwise, only a value of zero is supported.

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction leaf starts the launch of a measured environment by initiating a rendezvous sequence for all logical processors in the platform. The rendezvous sequence involves the initiating logical processor sending a message (by executing GETSEC[SENTER]) and other responding logical processors (RLPs) acknowledging the message, thus synchronizing the RLP(s) with the ILP.

In response to a message signaling the completion of rendezvous, RLPs clear the bootstrap processor indicator flag (IA32\_APIC\_BASE.BSP) and enter an SENTER sleep state. In this sleep state, RLPs enter an idle processor condition while waiting to be activated after a measured environment has been established by the system executive. RLPs in the SENTER sleep state can only be activated by the GETSEC leaf function WAKEUP in a measured environment.

A successful launch of the measured environment results in the initiating logical processor entering the authenticated code execution mode. Prior to reaching this point, the ILP performs the following steps internally:

- Inhibit processor response to the external events: INIT, A20M, NMI, and SMI.



- Establish and check the location and size of the authenticated code module to be executed by the ILP.
- Check for the existence of an Intel® TXT-capable chipset.
- Verify the current power management configuration is acceptable.
- Broadcast a message to enable protection of memory and I/O from activities from other processor agents.
- Load the designated AC module into authenticated code execution area.
- Isolate the content of authenticated code execution area from further state modification by external agents.
- Authenticate the AC module.
- Updated the Trusted Platform Module (TPM) with the authenticated code module's hash.
- Initialize processor state based on the authenticated code module header information.
- Unlock the Intel® TXT-capable chipset private configuration register space and TPM locality 3 space.
- Begin execution in the authenticated code module at the defined entry point.

As an integrity check for proper processor hardware operation, execution of GETSEC[SENDER] will also check the contents of all the machine check status registers (as reported by the MSRs IA32\_MCI\_STATUS) for any valid uncorrectable error condition. In addition, the global machine check status register IA32\_MCG\_STATUS MCIP bit must be cleared and the IERR processor package pin (or its equivalent) must be not asserted, indicating that no machine check exception processing is currently in-progress. These checks are performed twice: once by the ILP prior to the broadcast of the rendezvous message to RLPs, and later in response to RLPs acknowledging the rendezvous message. Any outstanding valid uncorrectable machine check error condition present in the machine check status registers at the first check point will result in the ILP signaling a general protection violation. If an outstanding valid uncorrectable machine check error condition is present at the second check point, then this will result in the corresponding logical processor signaling the more severe TXT-shutdown condition with an error code of 12.

Before loading and authentication of the target code module is performed, the processor also checks that the current voltage and bus ratio encodings correspond to known good values supportable by the processor. The MSR IA32\_PERF\_STATUS values are compared against either the processor supported maximum operating target setting, system reset setting, or the thermal monitor operating target. If the current settings do not meet any of these criteria then the SENTER function will attempt to change the voltage and bus ratio select controls in a processor-specific manner. This adjustment may be to the thermal monitor, minimum (if different), or maximum operating target depending on the processor.

This implies that some thermal operating target parameters configured by BIOS may be overridden by SENTER. The measured environment software may need to take responsibility for restoring such settings that are deemed to be safe, but not necessarily recognized by SENTER. If an adjustment is not possible when an out of range setting is discovered, then the processor will abort the measured launch. This may be the case for chipset controlled settings of these values or if the controllability is not enabled on the processor. In this case it is the responsibility of the external software to program the chipset voltage ID and/or bus ratio select settings to known good values recognized by the processor, prior to executing SENTER.

## NOTE

For a mobile processor, an adjustment can be made according to the thermal monitor operating target. For a quad-core processor the SENTER adjustment mechanism may result in a more conservative but non-uniform voltage setting, depending on the pre-SENDER settings per core.

The ILP and RLPs mask the response to the assertion of the external signals INIT#, A20M, NMI#, and SMI#. The purpose of this masking control is to prevent exposure to existing external event handlers until a protected handler has been put in place to directly handle these events. Masked external pin events may be unmasked conditionally or unconditionally via the GETSEC[EXITAC], GETSEC[SEXIT], GETSEC[SMCTRL] or for specific VMX related operations such as a VM entry or the VMXOFF instruction (see respective GETSEC leaves and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C* for more details). The state of the A20M pin is masked and forced internally to a de-asserted state so that external assertion is not recognized. A20M masking as set by GETSEC[SENDER] is undone only after taking down the measured environment with the GETSEC[SEXIT] instruction or processor reset. INTR is masked by simply clearing the EFLAGS.IF bit. It is the responsibility of system software to control the processor response to INTR through appropriate management of EFLAGS.

To prevent other (logical) processors from interfering with the ILP operating in authenticated code execution mode, memory (excluding implicit write-back transactions) and I/O activities originating from other processor agents are blocked. This protection starts when the ILP enters into authenticated code execution mode. Only memory and I/O transactions initiated from the ILP are allowed to proceed. Exiting authenticated code execution mode is done by executing GETSEC[EXITAC]. The protection of memory and I/O activities remains in effect until the ILP executes GETSEC[EXITAC].

Once the authenticated code module has been loaded into the authenticated code execution area, it is protected against further modification from external bus snoops. There is also a requirement that the memory type for the authenticated code module address range be WB (via initialization of the MTRRs prior to execution of this instruction). If this condition is not satisfied, it is a violation of security and the processor will force a TXT system reset (after writing an error code to the chipset LT.ERRORCODE register). This action is referred to as a Intel® TXT reset condition. It is performed when it is considered unreliable to signal an error through the conventional exception reporting mechanism.

To conform to the minimum granularity of MTRR MSRs for specifying the memory type, authenticated code RAM (ACRAM) is allocated to the processor in 4096 byte granular blocks. If an AC module size as specified in ECX is not a multiple of 4096 then the processor will allocate up to the next 4096 byte boundary for mapping as ACRAM with indeterminate data. This pad area will not be visible to the authenticated code module as external memory nor can it depend on the value of the data used to fill the pad area.

Once successful authentication has been completed by the ILP, the computed hash is stored in a trusted storage facility in the platform. The following trusted storage facility are supported:

- If the platform register FTM\_INTERFACE\_ID.[bits 3:0] = 0, the computed hash is stored to the platform's TPM at PCR17 after this register is implicitly reset. PCR17 is a dedicated register for holding the computed hash of the authenticated code module loaded and subsequently executed by the GETSEC[SENDER]. As part of this process, the dynamic PCRs 18-22 are reset so they can be utilized by subsequently software for registration of code and data modules.
- If the platform register FTM\_INTERFACE\_ID.[bits 3:0] = 1, the computed hash is stored in a firmware trusted module (FTM) using a modified protocol similar to the protocol used to write to TPM's PCR17.

After successful execution of SENTER, either PCR17 (if FTM is not enabled) or the FTM (if enabled) contains the measurement of AC code and the SENTER launching parameters.

After authentication is completed successfully, the private configuration space of the Intel® TXT-capable chipset is unlocked so that the authenticated code module and measured environment software can gain access to this normally restricted chipset state. The Intel® TXT-capable chipset private configuration space can be locked later by software writing to the chipset LT.CMD.CLOSE-PRIVATE register or unconditionally using the GETSEC[SEXIT] instruction.

The SENTER leaf function also initializes some processor architecture state for the ILP from contents held in the header of the authenticated code module. Since the authenticated code module is relocatable, all address references are relative to the base address passed in via EBX. The ILP GDTR base value is initialized to EBX + [GDTBasePtr] and GDTR limit set to [GDTLimit]. The CS selector is initialized to the value held in the AC module header field SegSel, while the DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE=0, LIMIT=FFFFh, G=1, D=1, P=1, S=1, read/write/accessed for DS, SS, and ES, while execute/read/accessed for CS. Execution in the authenticated code module for the ILP begins with the EIP set to EBX + [EntryPoint]. AC module defined fields used for initializing processor state are consistency checked with a failure resulting in an TXT-shutdown condition.

Table 6-6 provides a summary of processor state initialization for the ILP and RLP(s) after successful completion of GETSEC[SENDER]. For both ILP and RLP(s), paging is disabled upon entry to the measured environment. It is up to the ILP to establish a trusted paging environment, with appropriate mappings, to meet protection requirements established during the launch of the measured environment. RLP state initialization is not completed until a subsequent wake-up has been signaled by execution of the GETSEC[WAKEUP] function by the ILP.

**Table 6-6. Register State Initialization after GETSEC[SENDER] and GETSEC[WAKEUP]**

Register State	ILP after GETSEC[SENDER]	RLP after GETSEC[WAKEUP]
CR0	PG←0, AM←0, WP←0; Others unchanged	PG←0, CD←0, NW←0, AM←0, WP←0; PE←1, NE←1
CR4	00004000H	00004000H
EFLAGS	00000002H	00000002H
IA32_EFER	0H	0
EIP	[EntryPoint from MLE header <sup>1</sup> ]	[LT.MLE.JOIN + 12]
EBX	Unchanged [SINIT.BASE]	Unchanged
EDX	SENDER control flags	Unchanged
EBP	SINIT.BASE	Unchanged
CS	Sel=[SINIT.SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	Sel = [LT.MLE.JOIN + 8], base = 0, limit = FFFFFH, G = 1, D = 1, AR = 9BH
DS, ES, SS	Sel=[SINIT.SegSel] + 8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	Sel = [LT.MLE.JOIN + 8] + 8, base = 0, limit = FFFFFH, G = 1, D = 1, AR = 93H
GDTR	Base= SINIT.base (EBX) + [SINIT.GDTBasePtr], Limit=[SINIT.GDTLimit]	Base = [LT.MLE.JOIN + 4], Limit = [LT.MLE.JOIN]
DR7	00000400H	00000400H
IA32_DEBUGCTL	0H	0H
Performance counters and counter control registers	0H	0H
IA32_MISC_ENABLE	See Table 6-5	See Table 6-5
IA32_SMM_MONITOR_CTL	Bit 2←0	Bit 2←0

**NOTES:**

1. See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* for MLE header format.

Segmentation related processor state that has not been initialized by GETSEC[SENDER] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in FS, GS, TR, and LDTR may no longer be valid. The IDTR will also require reloading with a new IDT context after launching the measured environment before exceptions or the external interrupts INTR and NMI can be handled. In the meantime, the programmer must take care in not executing an INT n instruction or any other condition that would result in an exception or trap signaling.

Debug exception and trap related signaling is also disabled as part of execution of GETSEC[SENDER]. This is achieved by clearing DR7, TF in EFLAGS, and the MSR IA32\_DEBUGCTL as defined in Table 6-6. These can be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly re-initialized following SENTER. Also, any pending single-step trap condition will be cleared at the completion of SENTER for both the ILP and RLP(s).

Performance related counters and counter control registers are cleared as part of execution of SENTER on both the ILP and RLP. This implies any active performance counters at the time of SENTER execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

Since MCE along with all other state bits (with the exception of SMXE) are cleared in CR4 upon execution of SENTER processing, any enabled machine check error condition that occurs will result in the processor performing the TXT-

shutdown action. This also applies to an RLP while in the SENTER sleep state. For each logical processor CR4.MCE must be reestablished with a valid machine check exception handler to otherwise avoid an TXT-shutdown under such conditions.

The MSR IA32\_EFER is also unconditionally cleared as part of the processor state initialized by SENTER for both the ILP and RLP. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be re-established if it is desired to enable IA-32e mode while operating in authenticated code execution mode.

The miscellaneous feature control MSR, IA32\_MISC\_ENABLE, is initialized as part of the measured environment launch. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings. See the footnote for Table 6-5 The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. Among the impact of initializing this MSR, any previous condition established by the MONITOR instruction will be cleared.

#### Effect of MSR IA32\_FEATURE\_CONTROL MSR

Bits 15:8 of the IA32\_FEATURE\_CONTROL MSR affect the execution of GETSEC[SENTER]. These bits consist of two fields:

- Bit 15: a global enable control for execution of SENTER.
- Bits 14:8: a parameter control field providing the ability to qualify SENTER execution based on the level of functionality specified with corresponding EDX parameter bits 6:0.

The layout of these fields in the IA32\_FEATURE\_CONTROL MSR is shown in Table 6-1.

Prior to the execution of GETSEC[SENTER], the lock bit of IA32\_FEATURE\_CONTROL MSR must be bit set to affirm the settings to be used. Once the lock bit is set, only a power-up reset condition will clear this MSR. The IA32\_FEATURE\_CONTROL MSR must be configured in accordance to the intended usage at platform initialization. Note that this MSR is only available on SMX or VMX enabled processors. Otherwise, IA32\_FEATURE\_CONTROL is treated as reserved.

The *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* provides additional details and requirements for programming measured environment software to launch in an Intel TXT platform.

#### Operation in a Uni-Processor Platform

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

##### GETSEC[SENTER] (ILP only):

```
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((in VMX root operation) or
    (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or (TXT chipset not present) or
    (SENTERFLAG=1) or (ACMODEFLAG=1) or (IN_SMM=1) or
    (TPM interface is not present) or
    (EDX ≠ (SENTER_EDX_support_mask & EDX)) or
    (IA32_FEATURE_CONTROL[0]=0) or (IA32_FEATURE_CONTROL[15]=0) or
    ((IA32_FEATURE_CONTROL[14:8] & EDX[6:0]) ≠ EDX[6:0]))
    THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF IA32_MCG[I].STATUS = uncorrectable error
            THEN #GP(0);
    FI;
OD;
```

```

FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);
ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) ≠ 0) or ((ACSIZE MOD 64) ≠ 0) or (ACSIZE < minimum
    module size) or (ACSIZE > AC RAM capacity) or ((ACBASE+ACSIZE) > (2^32 -1)))
    THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
SignalTXTMsg(SENTER);
DO
WHILE (no SignalSENTER message);

```

#### **TXT\_SENTER\_\_MSG\_EVENT (ILP & RLP):**

```

Mask and clear SignalSENTER event;
Unmask SignalSEXIT event;
IF (in VMX operation)
    THEN TXT-SHUTDOWN(#IllegalEvent);
FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
    IF IA32_MC[I]_STATUS = uncorrectable error
        THEN TXT-SHUTDOWN(#UnrecovMCErr);
FI;
OD;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN TXT-SHUTDOWN(#UnrecovMCErr);
IF (Voltage or bus ratio status are NOT at a known good state)
    THEN IF (Voltage select and bus ratio are internally adjustable)
        THEN
            Make product-specific adjustment on operating parameters;
        ELSE
            TXT-SHUTDOWN(#IllegalVIDBRatio);
FI;

```

```

IA32_MISC_ENABLE← (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M← 0;
IA32_DEBUGCTL← 0;
Invalidate processor TLB(s);
Drain outgoing transactions;
Clear performance monitor counters and control;
SENTERFLAG← 1;
SignalTXTMsg(SENTERAck);
IF (logical processor is not ILP)
    THEN GOTO RLP_SENTER_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
    DONE← TXT.READ(LT.STS);
WHILE (not DONE);
SignalTXTMsg(SENTERContinue);
SignalTXTMsg(ProcessorHold);
FOR I=ACBASE to ACPBASE+ACSIZE-1 DO
    ACPRAM[I-ACBASE].ADDR← I;
    ACPRAM[I-ACBASE].DATA← LOAD(I);
OD;

```

```

IF (ACRAM memory type ≠ WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version is not supported) OR (ACRAM[ModuleType] ≠ 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
KEY ← GETKEY(ACRAM, ACBASE);
KEYHASH ← HASH(KEY);
CSKEYHASH ← LT.READ(LT.PUBLIC.KEY);
IF (KEYHASH ≠ CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE ← DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific *)
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I] ← SIGNATURE[I];
COMPUTEDSIGNATURE ← HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I] ← COMPUTEDSIGNATURE[I];
IF (SIGNATURE ≠ COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL ← ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified
    line detected on ACRAM load))
    THEN ACEntryPoint ← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint ← ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) or (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) or (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) or (ACRAM[SegSel].RPL≠0))
    THEN TXT-SHUTDOWN(#BadACMFormat);

IF (FTM_INTERFACE_ID.[3:0] = 1) (* Alternate FTM Interface has been enabled *)
    THEN (* TPM_LOC_CTRL_4 is located at 0FED44008H, TMP_DATA_BUFFER_4 is located at 0FED44080H *)
        WRITE(TPM_LOC_CTRL_4) ← 01H; (* Modified HASH.START protocol *)
        (* Write to firmware storage *)
        WRITE(TPM_DATA_BUFFER_4) ← SIGNATURE_LEN_CONST + 4;
        FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
            WRITE(TPM_DATA_BUFFER_4 + 2 + I) ← ACRAM[SCRATCH.I];
            WRITE(TPM_DATA_BUFFER_4 + 2 + SIGNATURE_LEN_CONST) ← EDX;
        WRITE(FTM.LOC_CTRL) ← 06H; (* Modified protocol combining HASH.DATA and HASH.END *)
    ELSE IF (FTM_INTERFACE_ID.[3:0] = 0) (* Use standard TPM Interface *)
        ACRAM[SCRATCH.SIGNATURE_LEN_CONST] ← EDX;
        WRITE(TPM.HASH.START) ← 0;
        FOR I=0 to SIGNATURE_LEN_CONST + 3 DO
            WRITE(TPM.HASH.DATA) ← ACRAM[SCRATCH.I];
        WRITE(TPM.HASH.END) ← 0;
FI;

```

```

ACMODEFLAG← 1;
CR0.[PG.AM.WP]← 0;
CR4← 00004000h;
EFLAGS← 00000002h;
IA32_EFER← 0;
EBP← ACBASE;
GDTR.BASE← ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT← ACRAM[GDTLimit];
CS.SEL← ACRAM[SegSel];
CS.BASE← 0;
CS.LIMIT← FFFFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← ACRAM[SegSel]+8;
DS.BASE← 0;
DS.LIMIT← FFFFFFFh;
DS.G← 1;
DS.D← 1;
DS.AR← 93h;
SS← DS;
ES← DS;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
SignalTXTMsg(UnlockSMRAM);
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP← ACEntryPoint;
END;

```

#### RLP\_SENTER\_ROUTINE: (RLP only)

```

Mask SMI, INIT, A20M, and NMI external pin events
Unmask SignalWAKEUP event;
Wait for SignalSENTERContinue message;
IA32_APIC_BASE.BSP← 0;
GOTO SENTER sleep state;
END;

```

#### Flags Affected

All flags are cleared.

#### Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

#### Protected Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[SENTER] is not reported as supported by GETSEC[CAPABILITIES].
-----	---

#GP(0) If CR0.CD = 1 or CR0.NW = 1 or CR0.NE = 0 or CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1.  
 If in VMX root operation.  
 If the initiating processor is not designated as the bootstrap processor via the MSR bit IA32\_APIC\_BASE.BSP.  
 If an Intel® TXT-capable chipset is not present.  
 If an Intel® TXT-capable chipset interface to TPM is not detected as present.  
 If a protected partition is already active or the processor is already in authenticated code mode.  
 If the processor is in SMM.  
 If a valid uncorrectable machine check error is logged in IA32\_MC[I]\_STATUS.  
 If the authenticated code base is not on a 4096 byte boundary.  
 If the authenticated code size > processor's authenticated code execution area storage capacity.  
 If the authenticated code size is not modulo 64.

### Real-Address Mode Exceptions

#UD If CR4.SMXE = 0.  
 If GETSEC[SENDER] is not reported as supported by GETSEC[CAPABILITIES].  
 #GP(0) GETSEC[SENDER] is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD If CR4.SMXE = 0.  
 If GETSEC[SENDER] is not reported as supported by GETSEC[CAPABILITIES].  
 #GP(0) GETSEC[SENDER] is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

All protected mode exceptions apply.

#GP IF AC code module does not reside in physical address below  $2^{32} - 1$ .

### 64-Bit Mode Exceptions

All protected mode exceptions apply.

#GP IF AC code module does not reside in physical address below  $2^{32} - 1$ .

### VM-Exit Condition

Reason (GETSEC) IF in VMX non-root operation.



## GETSEC[SEXIT]—Exit Measured Environment

Opcode	Instruction	Description
0F 37 (EAX=5)	GETSEC[SEXIT]	Exit measured environment.

### Description

The GETSEC[SEXIT] instruction initiates an exit of a measured environment established by GETSEC[SENDER]. The SEXIT leaf of GETSEC is selected with EAX set to 5 at execution. This instruction leaf sends a message to all logical processors in the platform to signal the measured environment exit.

There are restrictions enforced by the processor for the execution of the GETSEC[SEXIT] instruction:

- Execution is not allowed unless the processor is in protected mode (CR0.PE = 1) with CPL = 0 and EFLAGS.VM = 0.
- The processor must be in a measured environment as launched by a previous GETSEC[SENDER] instruction, but not still in authenticated code execution mode.
- To avoid potential inter-operability conflicts between modes, the processor is not allowed to execute this instruction if it currently is in SMM or in VMX operation.
- To insure consistent handling of SIPI messages, the processor executing the GETSEC[SEXIT] instruction must also be designated the BSP (bootstrap processor) as defined by the register bit IA32\_APIC\_BASE.BSP (bit 8).

Failure to abide by the above conditions results in the processor signaling a general protection violation.

This instruction initiates a sequence to rendezvous the RLPs with the ILP. It then clears the internal processor flag indicating the processor is operating in a measured environment.

In response to a message signaling the completion of rendezvous, all RLPs restart execution with the instruction that was to be executed at the time GETSEC[SEXIT] was recognized. This applies to all processor conditions, with the following exceptions:

- If an RLP executed HLT and was in this halt state at the time of the message initiated by GETSEC[SEXIT], then execution resumes in the halt state.
- If an RLP was executing MWAIT, then a message initiated by GETSEC[SEXIT] causes an exit of the MWAIT state, falling through to the next instruction.
- If an RLP was executing an intermediate iteration of a string instruction, then the processor resumes execution of the string instruction at the point which the message initiated by GETSEC[SEXIT] was recognized.
- If an RLP is still in the SENTER sleep state (never awakened with GETSEC[WAKEUP]), it will be sent to the wait-for-SIPI state after first clearing the bootstrap processor indicator flag (IA32\_APIC\_BASE.BSP) and any pending SIPI state. In this case, such RLPs are initialized to an architectural state consistent with having taken a soft reset using the INIT# pin.

Prior to completion of the GETSEC[SEXIT] operation, both the ILP and any active RLPs unmask the response of the external event signals INIT#, A20M, NMI#, and SMI#. This unmasking is performed unconditionally to recognize pin events which are masked after a GETSEC[SENDER]. The state of A20M is unmasked, as the A20M pin is not recognized while the measured environment is active.

On a successful exit of the measured environment, the ILP re-locks the Intel® TXT-capable chipset private configuration space. GETSEC[SEXIT] does not affect the content of any PCR.

At completion of GETSEC[SEXIT] by the ILP, execution proceeds to the next instruction. Since EFLAGS and the debug register state are not modified by this instruction, a pending trap condition is free to be signaled if previously enabled.

**Operation in a Uni-Processor Platform**

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

**GETSEC[SEXIT] (ILP only):**

```

IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((in VMX root operation) or
    (CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or
    (TXT chipset not present) or
    (SENTERFLAG=0) or (ACMODEFLAG=1) or (IN_SMM=1))
    THEN #GP(0);
SignalTXTMsg(SEXIT);
DO
WHILE (no SignalSEXIT message);

```

**TXT\_SEXIT\_MSG\_EVENT (ILP & RLP):**

```

Mask and clear SignalSEXIT event;
Clear MONITOR FSM;
Unmask SignalSENDER event;
IF (in VMX operation)
    THEN TXT-SHUTDOWN(#IllegalEvent);
SignalTXTMsg(SEXITAck);
IF (logical processor is not ILP)
    THEN GOTO RLP_SEXIT_ROUTINE;
(* ILP waits for all logical processors to ACK *)
DO
    DONE← READ(LT.STS);
WHILE (NOT DONE);
SignalTXTMsg(SEXITContinue);
SignalTXTMsg(ClosePrivate);
SENTERFLAG← 0;
Unmask SMI, INIT, A20M, and NMI external pin events;
END;

```

**RLP\_SEXIT\_ROUTINE (RLPs only):**

```

Wait for SignalSEXITContinue message;
Unmask SMI, INIT, A20M, and NMI external pin events;
IF (prior execution state = HLT)
    THEN reenter HLT state;
IF (prior execution state = SENTER sleep)
    THEN
        IA32_APIC_BASE.BSP← 0;
        Clear pending SIPI state;
        Call INIT_PROCESSOR_STATE;
        Unmask SIPI event;
        GOTO WAIT-FOR-SIPI;
FI;
END;

```

**Flags Affected**

ILP: None.

RLPs: all flags are modified for an RLP. returning to wait-for-SIPI state, none otherwise.

**Use of Prefixes**

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

**Protected Mode Exceptions**

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	<p>If CR0.PE = 0 or CPL &gt; 0 or EFLAGS.VM = 1.</p> <p>If in VMX root operation.</p> <p>If the initiating processor is not designated via the MSR bit IA32_APIC_BASE.BSP.</p> <p>If an Intel® TXT-capable chipset is not present.</p> <p>If a protected partition is not already active or the processor is already in authenticated code mode.</p> <p>If the processor is in SMM.</p>

**Real-Address Mode Exceptions**

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[SEXIT] is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[SEXIT] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[SEXIT] is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

All protected mode exceptions apply.

**64-Bit Mode Exceptions**

All protected mode exceptions apply.

**VM-Exit Condition**

Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[PARAMETERS]—Report the SMX Parameters

Opcode	Instruction	Description
0F 37 (EAX=6)	GETSEC[PARAMETERS]	Report the SMX parameters. The parameters index is input in EBX with the result returned in EAX, EBX, and ECX.

### Description

The GETSEC[PARAMETERS] instruction returns specific parameter information for SMX features supported by the processor. Parameter information is returned in EAX, EBX, and ECX, with the input parameter selected using EBX.

Software retrieves parameter information by searching with an input index for EBX starting at 0, and then reading the returned results in EAX, EBX, and ECX. EAX[4:0] is designated to return a parameter type field indicating if a parameter is available and what type it is. If EAX[4:0] is returned with 0, this designates a null parameter and indicates no more parameters are available.

Table 6-7 defines the parameter types supported in current and future implementations.

**Table 6-7. SMX Reporting Parameters Format**

Parameter Type EAX[4:0]	Parameter Description	EAX[31:5]	EBX[31:0]	ECX[31:0]
0	NULL	Reserved (0 returned)	Reserved (unmodified)	Reserved (unmodified)
1	Supported AC module versions	Reserved (0 returned)	Version comparison mask	Version numbers supported
2	Max size of authenticated code execution area	Multiply by 32 for size in bytes	Reserved (unmodified)	Reserved (unmodified)
3	External memory types supported during AC mode	Memory type bit mask	Reserved (unmodified)	Reserved (unmodified)
4	Selective SENTER functionality control	EAX[14:8] correspond to available SENTER function disable controls	Reserved (unmodified)	Reserved (unmodified)
5	TXT extensions support	TXT Feature Extensions Flags (see Table 6-8)	Reserved	Reserved
6-31	Undefined	Reserved (unmodified)	Reserved (unmodified)	Reserved (unmodified)

**Table 6-8. TXT Feature Extensions Flags**

Bit	Definition	Description
5	Processor based S-CRTM support	Returns 1 if this processor implements a processor-rooted S-CRTM capability and 0 if not (S-CRTM is rooted in BIOS). This flag cannot be used to infer whether the chipset supports TXT or whether the processor support SMX.
6	Machine Check Handling	Returns 1 if it machine check status registers can be preserved through ENTERACCS and SENTER. If this bit is 1, the caller of ENTERACCS and SENTER is not required to clear machine check error status bits before invoking these GETSEC leaves. If this bit returns 0, the caller of ENTERACCS and SENTER must clear all machine check error status bits before invoking these GETSEC leaves.
31:7	Reserved	Reserved for future use. Will return 0.

Supported AC module versions (as defined by the AC module HeaderVersion field) can be determined for a particular SMX capable processor by the type 1 parameter. Using EBX to index through the available parameters reported by GETSEC[PARAMETERS] for each unique parameter set returned for type 1, software can determine the complete list of AC module version(s) supported.

For each parameter set, EBX returns the comparison mask and ECX returns the available HeaderVersion field values supported, after AND'ing the target HeaderVersion with the comparison mask. Software can then determine if a particular AC module version is supported by following the pseudo-code search routine given below:

```
parameter_search_index= 0
do {
    EBX= parameter_search_index++
    EAX= 6
    GETSEC
    if (EAX[4:0] = 1) {
        if ((version_query & EBX) = ECX) {
            version_is_supported= 1
            break
        }
    }
} while (EAX[4:0] ≠ 0)
```

If only AC modules with a HeaderVersion of 0 are supported by the processor, then only one parameter set of type 1 will be returned, as follows: EAX = 00000001H,

EBX = FFFFFFFFH and ECX = 00000000H.

The maximum capacity for an authenticated code execution area supported by the processor is reported with the parameter type of 2. The maximum supported size in bytes is determined by multiplying the returned size in EAX[31:5] by 32. Thus, for a maximum supported authenticated RAM size of 32KBytes, EAX returns with 00008002H.

Supportable memory types for memory mapped outside of the authenticated code execution area are reported with the parameter type of 3. While is active, as initiated by the GETSEC functions SENTER and ENTERACCS and terminated by EXITAC, there are restrictions on what memory types are allowed for the rest of system memory. It is the responsibility of the system software to initialize the memory type range register (MTRR) MSRs and/or the page attribute table (PAT) to only map memory types consistent with the reporting of this parameter. The reporting of supportable memory types of external memory is indicated using a bit map returned in EAX[31:8]. These bit positions correspond to the memory type encodings defined for the MTRR MSR and PAT programming. See Table 6-9.

The parameter type of 4 is used for enumerating the availability of selective GETSEC[SENDER] function disable controls. If a 1 is reported in bits 14:8 of the returned parameter EAX, then this indicates a disable control capa-

bility exists with SENTER for a particular function. The enumerated field in bits 14:8 corresponds to use of the EDX input parameter bits 6:0 for SENTER. If an enumerated field bit is set to 1, then the corresponding EDX input parameter bit of EDX may be set to 1 to disable that designated function. If the enumerated field bit is 0 or this parameter is not reported, then no disable capability exists with the corresponding EDX input parameter for SENTER, and EDX bit(s) must be cleared to 0 to enable execution of SENTER. If no selective disable capability for SENTER exists as enumerated, then the corresponding bits in the IA32\_FEATURE\_CONTROL MSR bits 14:8 must also be programmed to 1 if the SENTER global enable bit 15 of the MSR is set. This is required to enable future extensibility of SENTER selective disable capability with respect to potentially separate software initialization of the MSR.

**Table 6-9. External Memory Types Using Parameter 3**

EAX Bit Position	Parameter Description
8	Uncacheable (UC)
9	Write Combining (WC)
11:10	Reserved
12	Write-through (WT)
13	Write-protected (WP)
14	Write-back (WB)
31:15	Reserved

If the GETSEC[PARAMETERS] leaf or specific parameter is not present for a given SMX capable processor, then default parameter values should be assumed. These are defined in Table 6-10.

**Table 6-10. Default Parameter Values**

Parameter Type EAX[4:0]	Default Setting	Parameter Description
1	0.0 only	Supported AC module versions.
2	32 KBytes	Authenticated code execution area size.
3	UC only	External memory types supported during AC execution mode.
4	None	Available SENTER selective disable controls.

## Operation

(\* example of a processor supporting only a 0.0 HeaderVersion, 32K ACRAM size, memory types UC and WC \*)

IF (CR4.SMXE=0)

THEN #UD;

ELSE IF (in VMX non-root operation)

THEN VM Exit (reason="GETSEC instruction");

ELSE IF (GETSEC leaf unsupported)

THEN #UD;

(\* example of a processor supporting a 0.0 HeaderVersion \*)

IF (EBX=0) THEN

EAX← 00000001h;

EBX← FFFFFFFFh;

ECX← 00000000h;

ELSE IF (EBX=1)

(\* example of a processor supporting a 32K ACRAM size \*)

```
    THEN EAX← 00008002h;
ESE IF (EBX= 2)
    (* example of a processor supporting external memory types of UC and WC *)
    THEN EAX← 00000303h;
ESE IF (EBX= other value(s) less than unsupported index value)
    (* EAX value varies. Consult Table 6-7 and Table 6-8*)
ELSE (* unsupported index*)
    EAX" 00000000h;
END;
```

Flags Affected

None.

Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

Protected Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].
-----	---

Real-Address Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].
-----	---

Virtual-8086 Mode Exceptions

#UD	If CR4.SMXE = 0. If GETSEC[PARAMETERS] is not reported as supported by GETSEC[CAPABILITIES].
-----	---

Compatibility Mode Exceptions

All protected mode exceptions apply.

64-Bit Mode Exceptions

All protected mode exceptions apply.

VM-Exit Condition

Reason (GETSEC) IF in VMX non-root operation.

## GETSEC[SMCTRL]—SMX Mode Control

Opcode	Instruction	Description
OF 37 (EAX = 7)	GETSEC[SMCTRL]	Perform specified SMX mode control as selected with the input EBX.

### Description

The GETSEC[SMCTRL] instruction is available for performing certain SMX specific mode control operations. The operation to be performed is selected through the input register EBX. Currently only an input value in EBX of 0 is supported. All other EBX settings will result in the signaling of a general protection violation.

If EBX is set to 0, then the SMCTRL leaf is used to re-enable SMI events. SMI is masked by the ILP executing the GETSEC[SENTER] instruction (SMI is also masked in the responding logical processors in response to SENTER rendezvous messages.). The determination of when this instruction is allowed and the events that are unmasked is dependent on the processor context (See Table 6-11). For brevity, the usage of SMCTRL where EBX=0 will be referred to as GETSEC[SMCTRL(0)].

As part of support for launching a measured environment, the SMI, NMI and INIT events are masked after GETSEC[SENTER], and remain masked after exiting authenticated execution mode. Unmasking these events should be accompanied by securely enabling these event handlers. These security concerns can be addressed in VMX operation by a MVMM.

The VM monitor can choose two approaches:

- In a dual monitor approach, the executive software will set up an SMM monitor in parallel to the executive VMM (i.e. the MVMM), see Chapter 34, “System Management Mode” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*. The SMM monitor is dedicated to handling SMI events without compromising the security of the MVMM. This usage model of handling SMI while a measured environment is active does not require the use of GETSEC[SMCTRL(0)] as event re-enabling after the VMX environment launch is handled implicitly and through separate VMX based controls.
- If a dedicated SMM monitor will not be established and SMIs are to be handled within the measured environment, then GETSEC[SMCTRL(0)] can be used by the executive software to re-enable SMI that has been masked as a result of SENTER.

Table 6-11 defines the processor context in which GETSEC[SMCTRL(0)] can be used and which events will be unmasked. Note that the events that are unmasked are dependent upon the currently operating processor context.

**Table 6-11. Supported Actions for GETSEC[SMCTRL(0)]**

ILP Mode of Operation	SMCTRL execution action
In VMX non-root operation	VM exit
SENTERFLAG = 0	#GP(0), illegal context
In authenticated code execution mode (ACMODEFLAG = 1)	#GP(0), illegal context
SENTERFLAG = 1, not in VMX operation, not in SMM	Unmask SMI
SENTERFLAG = 1, in VMX root operation, not in SMM	Unmask SMI if SMM monitor is not configured, otherwise #GP(0)
SENTERFLAG = 1, In VMX root operation, in SMM	#GP(0), illegal context



**Operation**

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

```
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) OR (EFLAGS.VM=1))
    THEN #GP(0);
ELSE IF ((EBX=0) and (SENTERFLAG=1) and (ACMODEFLAG=0) and (IN_SMM=0) and
    (((in VMX root operation) and (SMM monitor not configured)) or (not in VMX operation)))
    THEN unmask SMI;
ELSE
    #GP(0);
END
```

**Flags Affected**

None.

**Use of Prefixes**

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

**Protected Mode Exceptions**

#UD	If CR4.SMXE = 0. If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	If CR0.PE = 0 or CPL > 0 or EFLAGS.VM = 1. If in VMX root operation. If a protected partition is not already active or the processor is currently in authenticated code mode. If the processor is in SMM. If the SMM monitor is not configured.

**Real-Address Mode Exceptions**

#UD	If CR4.SMXE = 0. If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SMCTRL] is not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD	If CR4.SMXE = 0. If GETSEC[SMCTRL] is not reported as supported by GETSEC[CAPABILITIES].
#GP(0)	GETSEC[SMCTRL] is not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

All protected mode exceptions apply.

**64-Bit Mode Exceptions**

All protected mode exceptions apply.

**VM-exit Condition**

Reason (GETSEC) IF in VMX non-root operation.

GETSEC[WAKEUP]—Wake up sleeping processors in measured environment

Opcode	Instruction	Description
OF 37 (EAX=8)	GETSEC[WAKEUP]	Wake up the responding logical processors from the SENTER sleep state.

Description

The GETSEC[WAKEUP] leaf function broadcasts a wake-up message to all logical processors currently in the SENTER sleep state. This GETSEC leaf must be executed only by the ILP, in order to wake-up the RLPs. Responding logical processors (RLPs) enter the SENTER sleep state after completion of the SENTER rendezvous sequence.

The GETSEC[WAKEUP] instruction may only be executed:

- In a measured environment as initiated by execution of GETSEC[SENTER].
- Outside of authenticated code execution mode.
- Execution is not allowed unless the processor is in protected mode with CPL = 0 and EFLAGS.VM = 0.
- In addition, the logical processor must be designated as the boot-strap processor as configured by setting IA32\_APIC\_BASE.BSP = 1.

If these conditions are not met, attempts to execute GETSEC[WAKEUP] result in a general protection violation.

An RLP exits the SENTER sleep state and start execution in response to a WAKEUP signal initiated by ILP’s execution of GETSEC[WAKEUP]. The RLP retrieves a pointer to a data structure that contains information to enable execution from a defined entry point. This data structure is located using a physical address held in the Intel® TXT-capable chipset configuration register LT.MLE.JOIN. The register is publicly writable in the chipset by all processors and is not restricted by the Intel® TXT-capable chipset configuration register lock status. The format of this data structure is defined in Table 6-12.

Table 6-12. RLP MVM JOIN Data Structure

Offset	Field
0	GDT limit
4	GDT base pointer
8	Segment selector initializer
12	EIP

The MLE JOIN data structure contains the information necessary to initialize RLP processor state and permit the processor to join the measured environment. The GDTR, LIP, and CS, DS, SS, and ES selector values are initialized using this data structure. The CS selector index is derived directly from the segment selector initializer field; DS, SS, and ES selectors are initialized to CS+8. The segment descriptor fields are initialized implicitly with BASE = 0, LIMIT = FFFFFFFH, G = 1, D = 1, P = 1, S = 1; read/write/access for DS, SS, and ES; and execute/read/access for CS. It is the responsibility of external software to establish a GDT pointed to by the MLE JOIN data structure that contains descriptor entries consistent with the implicit settings initialized by the processor (see Table 6-6). Certain states from the content of Table 6-12 are checked for consistency by the processor prior to execution. A failure of any consistency check results in the RLP aborting entry into the protected environment and signaling an Intel® TXT shutdown condition. The specific checks performed are documented later in this section. After successful completion of processor consistency checks and subsequent initialization, RLP execution in the measured environment begins from the entry point at offset 12 (as indicated in Table 6-12).

**Operation**

(\* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary \*)

```

IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or (SENTERFLAG=0) or (ACMODEFLAG=1) or (IN_SMM=0) or (in VMX operation) or
(IA32_APIC_BASE.BSP=0) or (TXT chipset not present))
    THEN #GP(0);
ELSE
    SignalTXTMsg(WAKEUP);
END;

```

**RLP\_SIPWAKEUP\_FROM\_SENTER\_ROUTINE: (RLP only)**

```

WHILE (no SignalWAKEUP event);
IF (IA32_SMM_MONITOR_CTL[0] ≠ ILP.IA32_SMM_MONITOR_CTL[0])
    THEN TXT-SHUTDOWN(#IllegalEvent)
IF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ELSE
    Mask SMI pin event;
Mask A20M, and NMI external pin events (unmask INIT);
Mask SignalWAKEUP event;
Invalidate processor TLB(s);
Drain outgoing transactions;
TempGDTRLIMIT← LOAD(LT.MLE JOIN);
TempGDTRBASE← LOAD(LT.MLE JOIN+4);
TempSegSel← LOAD(LT.MLE JOIN+8);
TempEIP← LOAD(LT.MLE JOIN+12);
IF (TempGDTLimit & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel > TempGDTRLIMIT-15) or (TempSegSel < 8))
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel.TI=1) or (TempSegSel.RPL≠0))
    THEN TXT-SHUTDOWN(#BadJOINFormat);
CR0.[PG,CD,NW,AM,WP]← 0;
CR0.[NE,PE]← 1;
CR4← 00004000h;
EFLAGS← 00000002h;
IA32_EFER← 0;
GDTR.BASE← TempGDTRBASE;
GDTR.LIMIT← TempGDTRLIMIT;
CS.SEL← TempSegSel;
CS.BASE← 0;
CS.LIMIT← FFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← TempSegSel+8;
DS.BASE← 0;
DS.LIMIT← FFFFh;
DS.G← 1;

```

```

DS.D ← 1;
DS.AR ← 93h;
SS ← DS;
ES ← DS;
DR7 ← 00000400h;
IA32_DEBUGCTL ← 0;
EIP ← TempEIP;
END;

```

### Flags Affected

None.

### Use of Prefixes

LOCK	Causes #UD.
REP*	Cause #UD (includes REPNE/REPNZ and REP/REPE/REPZ).
Operand size	Causes #UD.
Segment overrides	Ignored.
Address size	Ignored.
REX	Ignored.

### Protected Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	<p>If CR0.PE = 0 or CPL &gt; 0 or EFLAGS.VM = 1.</p> <p>If in VMX operation.</p> <p>If a protected partition is not already active or the processor is currently in authenticated code mode.</p> <p>If the processor is in SMM.</p>
#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[WAKEUP] is not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

#UD	<p>If CR4.SMXE = 0.</p> <p>If GETSEC[WAKEUP] is not reported as supported by GETSEC[CAPABILITIES].</p>
#GP(0)	GETSEC[WAKEUP] is not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

All protected mode exceptions apply.

### 64-Bit Mode Exceptions

All protected mode exceptions apply.

### VM-exit Condition

Reason (GETSEC) IF in VMX non-root operation.

## PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
OF 0D /2 PREFETCHWT1 m8	M	V/V	PREFETCHWT1	Move data from m8 closer to the processor using T1 hint with intent to write.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by an intent to write hint (so that data is brought into 'Exclusive' state via a request for ownership) and a locality hint:

- T1 (temporal data with respect to first level cache)—prefetch data into the second level cache.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHH instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHH instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHH instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHH instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHH instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR. This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### Operation

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

Prefetch (m8, Level = 1, EXCLUSIVE=1);

### Flags Affected

All flags are affected

### C/C++ Compiler Intrinsic Equivalent

```
void _mm_prefetch( char const *, int hint= _MM_HINT_ET1);
```

### Protected Mode Exceptions

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD                      If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#UD                      If the LOCK prefix is used.

**Compatibility Mode Exceptions**

#UD                      If the LOCK prefix is used.

**64-Bit Mode Exceptions**

#UD                      If the LOCK prefix is used.

## VEXP2PD—Approximation to the Exponential $2^x$ of Packed Double-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 C8 /r VEXP2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

### Description

Computes the approximate base-2 exponential evaluation of the double-precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than  $2^{-23}$  of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VEXP2xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VEXP2PD

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

  i ← j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i] ← EXP2\_23\_DP(SRC[63:0])

      ELSE DEST[i+63:i] ← EXP2\_23\_DP(SRC[i+63:i])

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

    FI;

  FI;

ENDFOR;



**Table 6-33. Special Values Behavior**

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
$+\infty$	$+\infty$	
$\pm 0$	1.0f	<i>Exact result</i>
$-\infty$	$+\infty$	
Integral value N	$2^N$	<i>Exact result</i>

**Intel C/C++ Compiler Intrinsic Equivalent**

VEXP2PD \_\_m512d \_\_mm512\_exp2a23\_round\_pd (\_\_m512d a, int sae);

VEXP2PD \_\_m512d \_\_mm512\_mask\_exp2a23\_round\_pd (\_\_m512d a, \_\_mmask8 m, \_\_m512d b, int sae);

VEXP2PD \_\_m512d \_\_mm512\_maskz\_exp2a23\_round\_pd (\_\_mmask8 m, \_\_m512d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Overflow

**Other Exceptions**

See Exceptions Type E2.

## VEXP2PS—Approximation to the Exponential $2^x$ of Packed Single-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C8 /r VEXP2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

### Description

Computes the approximate base-2 exponential evaluation of the single-precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than  $2^{-23}$  of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VEXP2xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VEXP2PS

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+31:i] ← EXP2\_23\_SP(SRC[31:0])

      ELSE DEST[i+31:i] ← EXP2\_23\_SP(SRC[i+31:i])

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

    FI;

  FI;

ENDFOR;

**Table 6-34. Special Values Behavior**

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
$+\infty$	$+\infty$	
$\pm 0$	1.0f	<i>Exact result</i>
$-\infty$	$+\infty$	
Integral value N	$2^N$	<i>Exact result</i>

**Intel C/C++ Compiler Intrinsic Equivalent**

VEXP2PS \_\_m512 \_\_mm512\_exp2a23\_round\_ps (\_\_m512 a, int sae);

VEXP2PS \_\_m512 \_\_mm512\_mask\_exp2a23\_round\_ps (\_\_m512 a, \_\_mmask16 m, \_\_m512 b, int sae);

VEXP2PS \_\_m512 \_\_mm512\_maskz\_exp2a23\_round\_ps (\_\_mmask16 m, \_\_m512 b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Overflow

**Other Exceptions**

See Exceptions Type E2.

## VGATHERPFODPS/VGATHERPFOQPS/VGATHERPFODPD/VGATHERPFOQPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /1 /vsib VGATHERPFODPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W0 C7 /1 /vsib VGATHERPFOQPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C6 /1 /vsib VGATHERPFODPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C7 /1 /vsib VGATHERPFOQPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

**VGATHERPFODPS (EVEX encoded version)**

```

(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+31:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

```

**VGATHERPFODPD (EVEX encoded version)**

```

(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

```

**VGATHERPFOQPS (EVEX encoded version)**

```

(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[j+63:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

```

**VGATHERPFOQPD (EVEX encoded version)**

```

(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VGATHERPFODPD void __mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPFODPS void __mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPFOQPD void __mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPFOQPS void __mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.

## VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /2 /vsib VGATHERPF1DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W0 C7 /2 /vsib VGATHERPF1QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C6 /2 /vsib VGATHERPF1DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C7 /2 /vsib VGATHERPF1QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

**VGATHERPF1DPS (EVEX encoded version)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[j+31:i]) \* SCALE + DISP], Level=1, RFO = 0)

FI;

ENDFOR

**VGATHERPF1DPD (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP], Level=1, RFO = 0)

FI;

ENDFOR

**VGATHERPF1QPS (EVEX encoded version)**

(KL, VL) = (8, 256)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[j+63:i]) \* SCALE + DISP], Level=1, RFO = 0)

FI;

ENDFOR

**VGATHERPF1QPD (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 64

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[k+63:k]) \* SCALE + DISP], Level=1, RFO = 0)

FI;

ENDFOR

**Intel C/C++ Compiler Intrinsic Equivalent**

VGATHERPF1DPD void \_\_mm512\_mask\_prefetch\_i32gather\_pd(\_\_m256i vdx, \_\_mmask8 m, void \* base, int scale, int hint);

VGATHERPF1DPS void \_\_mm512\_mask\_prefetch\_i32gather\_ps(\_\_m512i vdx, \_\_mmask16 m, void \* base, int scale, int hint);

VGATHERPF1QPD void \_\_mm512\_mask\_prefetch\_i64gather\_pd(\_\_m512i vdx, \_\_mmask8 m, void \* base, int scale, int hint);

VGATHERPF1QPS void \_\_mm512\_mask\_prefetch\_i64gather\_ps(\_\_m512i vdx, \_\_mmask8 m, void \* base, int scale, int hint);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.

## VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CA /r VRCP28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes the approximate reciprocals ( $< 2^{-28}$ relative error) of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRCP28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vcvtp14-vrsqrt14-vcvtp28-vrsqrt28-vexp2>.

### Operation

#### VRCP28PD (EVEX encoded versions)

(KL, VL) = (8, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i]  $\leftarrow$  RCP\_28\_DP(1.0/SRC[63:0]);

      ELSE DEST[i+63:i]  $\leftarrow$  RCP\_28\_DP(1.0/SRC[i+63:i]);

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;



**Table 6-35. VRCP28PD Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \leq X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \leq -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	$2^n$	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	$-2^n$	Exact result (unless input/output is a denormal)

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP28PD \_\_m512d \_\_mm512\_rcp28\_round\_pd ( \_\_m512d a, int sae);

VRCP28PD \_\_m512d \_\_mm512\_mask\_rcp28\_round\_pd(\_\_m512d a, \_\_mmask8 m, \_\_m512d b, int sae);

VRCP28PD \_\_m512d \_\_mm512\_maskz\_rcp28\_round\_pd( \_\_mmask8 m, \_\_m512d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

## VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 CB /r VRCP28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	A	V/V	AVX512ER	Computes the approximate reciprocal ( $< 2^{-28}$ relative error) of the scalar double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP28SD ((EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[63: 0] ← RCP_28_DP(1.0/SRC2[63: 0]);
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[63: 0] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[63: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:64] ← SRC1[127: 64]
DEST[MAXVL-1:128] ← 0

```

**Table 6-36. VRCP28SD Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \leq X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \leq -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	$2^n$	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	$-2^n$	Exact result (unless input/output is a denormal)

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP28SD \_\_m128d \_\_mm\_rcp28\_round\_sd ( \_\_m128d a, \_\_m128d b, int sae);

VRCP28SD \_\_m128d \_\_mm\_mask\_rcp28\_round\_sd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128d b, int sae);

VRCP28SD \_\_m128d \_\_mm\_maskz\_rcp28\_round\_sd(\_\_mmask8 m, \_\_m128d a, \_\_m128d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CA /r VRCP28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes the approximate reciprocals ( $< 2^{-28}$ relative error) of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final results are rounded to  $< 2^{-23}$  relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRCP28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP28PS (EVEX encoded versions)

(KL, VL) = (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+31:i]  $\leftarrow$  RCP\_28\_SP(1.0/SRC[31:0]);

      ELSE DEST[i+31:i]  $\leftarrow$  RCP\_28\_SP(1.0/SRC[i+31:i]);

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

**Table 6-37. VRCP28PS Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \leq X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \leq -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	$2^n$	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	$-2^n$	Exact result (unless input/output is a denormal)

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP28PS \_\_mm512\_rcp28\_round\_ps ( \_\_m512 a, int sae);

VRCP28PS \_\_m512 \_\_mm512\_mask\_rcp28\_round\_ps(\_\_m512 s, \_\_mmask16 m, \_\_m512 a, int sae);

VRCP28PS \_\_m512 \_\_mm512\_maskz\_rcp28\_round\_ps( \_\_mmask16 m, \_\_m512 a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

## VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 CB /r VRCP28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	A	V/V	AVX512ER	Computes the approximate reciprocal ( $< 2^{-28}$ relative error) of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single-precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

### Description

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final result is rounded to  $< 2^{-23}$  relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FTZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is  $\pm\infty$ ,  $\pm 0.0$  is returned for that element. Also, if any source element is  $\pm 0.0$ ,  $\pm\infty$  is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRCP28SS ((EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[31: 0] ← RCP_28_SP(1.0/SRC2[31: 0]);
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[31: 0] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[31: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:32] ← SRC1[127: 32]
DEST[MAXVL-1:128] ← 0

```

**Table 6-38. VRCP28SS Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \leq X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \leq -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	$2^n$	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	$-2^n$	Exact result (unless input/output is a denormal)

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP28SS \_\_m128\_mm\_rcp28\_round\_ss ( \_\_m128 a, \_\_m128 b, int sae);

VRCP28SS \_\_m128\_mm\_mask\_rcp28\_round\_ss(\_\_m128 s, \_\_mmask8 m, \_\_m128 a, \_\_m128 b, int sae);

VRCP28SS \_\_m128\_mm\_maskz\_rcp28\_round\_ss(\_\_mmask8 m, \_\_m128 a, \_\_m128 b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CC /r VRSQRT28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes approximations to the Reciprocal square root ( $<2^{-28}$ relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRSQRT28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT28PD (EVEX encoded versions)

(KL, VL) = (8, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i]  $\leftarrow$  (1.0/ SQRT(SRC[63:0]));

      ELSE DEST[i+63:i]  $\leftarrow$  (1.0/ SQRT(SRC[i+63:i]));

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;



**Table 6-39. VRSQRT28PD Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNAN_Indefinite	Including -INF
$X = -0$ or negative denormal	-INF	#Z
$X = +0$ or positive denormal	+INF	#Z
$X = +\text{INF}$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28PD \_\_m512d \_\_mm512\_rsqrt28\_round\_pd(\_\_m512d a, int sae);

VRSQRT28PD \_\_m512d \_\_mm512\_mask\_rsqrt28\_round\_pd(\_\_m512d s, \_\_mmask8 m, \_\_m512d a, int sae);

VRSQRT28PD \_\_m512d \_\_mm512\_maskz\_rsqrt28\_round\_pd(\_\_mmask8 m, \_\_m512d a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

## VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 CD /r VRSQRT28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root ( $<2^{-28}$ relative error) of the scalar double-precision floating-point value from xmm3/m64 and stores result in xmm1 with writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than  $2^{-28}$  of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT28SD (EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[63: 0] ← (1.0/ SQRT(SRC[63: 0]));
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[63: 0] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[63: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:64] ← SRC1[127: 64]
DEST[MAXVL-1:128] ← 0

```

**Table 6-40. VRSQRT28SD Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNAN_Indefinite	Including -INF
$X = -0$ or negative denormal	-INF	#Z
$X = +0$ or positive denormal	+INF	#Z
$X = +\text{INF}$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28SD \_\_m128d \_\_mm\_rsqrt28\_round\_sd(\_\_m128d a, \_\_m128b b, int sae);

VRSQRT28SD \_\_m128d \_\_mm\_mask\_rsqrt28\_round\_pd(\_\_m128d s, \_\_mmask8 m, \_\_m128d a, \_\_m128d b, int sae);

VRSQRT28SD \_\_m128d \_\_mm\_maskz\_rsqrt28\_round\_pd(\_\_mmask8 m, \_\_m128d a, \_\_m128d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CC /r VRSQRT28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes approximations to the Reciprocal square root ( $<2^{-28}$ relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores result in zmm1 with writemask k1.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final results is rounded to  $< 2^{-23}$  relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRSQRT28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT28PS (EVEX encoded versions)

(KL, VL) = (16, 512)

FOR j  $\leftarrow$  0 TO KL-1

  i  $\leftarrow$  j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC \*is memory\*)

      THEN DEST[i+31:i]  $\leftarrow$  (1.0/ SQRT(SRC[31:0]));

      ELSE DEST[i+31:i]  $\leftarrow$  (1.0/ SQRT(SRC[i+31:i]));

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+31:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+31:i]  $\leftarrow$  0

    FI;

  FI;

ENDFOR;

**Table 6-41. VRSQRT28PS Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNAN_Indefinite	Including -INF
$X = -0$ or negative denormal	-INF	#Z
$X = +0$ or positive denormal	+INF	#Z
$X = +\text{INF}$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28PS \_\_m512 \_mm512\_rsqrt28\_round\_ps(\_\_m512 a, int sae);

VRSQRT28PS \_\_m512 \_mm512\_mask\_rsqrt28\_round\_ps(\_\_m512 s, \_\_mmask16 m, \_\_m512 a, int sae);

VRSQRT28PS \_\_m512 \_mm512\_maskz\_rsqrt28\_round\_ps(\_\_mmask16 m, \_\_m512 a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.

## VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 CD /r VRSQRT28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root ( $<2^{-28}$ relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1 with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than  $2^{-28}$  of maximum relative error prior to final rounding. The final result is rounded to  $< 2^{-23}$  relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as  $-\infty$ , return the canonical NaN and set the Invalid Flag (#I).

A value of  $-0$  must return  $-\infty$  and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return  $-\infty$  and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT28SS (EVEX encoded versions)

```

IF k1[0] OR *no writemask* THEN
    DEST[31:0] ← (1.0/ SQRT(SRC[31:0]));
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[31:0] remains unchanged*
    ELSE                                ; zeroing-masking
        DEST[31:0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

```

**Table 6-42. VRSQRT28SS Special Cases**

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	$2^n$	
$X < 0$	QNAN_Indefinite	Including -INF
$X = -0$ or negative denormal	-INF	#Z
$X = +0$ or positive denormal	+INF	#Z
$X = +\text{INF}$	+0	

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRT28SS \_\_m128\_mm\_rsqrt28\_round\_ss(\_\_m128 a, \_\_m128 b, int sae);

VRSQRT28SS \_\_m128\_mm512\_mask\_rsqrt28\_round\_ss(\_\_m128 s, \_\_mmask8 m, \_\_m128 a, \_\_m128 b, int sae);

VRSQRT28SS \_\_m128\_mm512\_maskz\_rsqrt28\_round\_ss(\_\_mmask8 m, \_\_m128 a, \_\_m128 b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.

## VSCATTERPFODPS/VSCATTERPFOQPS/VSCATTERPFODPD/VSCATTERPFOQPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint with Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /5 /vsib VSCATTERPFODPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /5 /vsib VSCATTERPFOQPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /5 /vsib VSCATTERPFODPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /5 /vsib VSCATTERPFOQPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.



**VSCATTERPFODPS (EVEX encoded version)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[j+31:i]) \* SCALE + DISP], Level=0, RFO = 1)

FI;

ENDFOR

**VSCATTERPFODPD (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP], Level=0, RFO = 1)

FI;

ENDFOR

**VSCATTERPFOQPS (EVEX encoded version)**

(KL, VL) = (8, 256)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[j+63:i]) \* SCALE + DISP], Level=0, RFO = 1)

FI;

ENDFOR

**VSCATTERPFOQPD (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 64

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[k+63:k]) \* SCALE + DISP], Level=0, RFO = 1)

FI;

ENDFOR

**Intel C/C++ Compiler Intrinsic Equivalent**

VSCATTERPFODPD void \_\_mm512\_prefetch\_i32scatter\_pd(void \*base, \_\_m256i vdx, int scale, int hint);

VSCATTERPFODPD void \_\_mm512\_mask\_prefetch\_i32scatter\_pd(void \*base, \_\_mmask8 m, \_\_m256i vdx, int scale, int hint);

VSCATTERPFODPS void \_\_mm512\_prefetch\_i32scatter\_ps(void \*base, \_\_m512i vdx, int scale, int hint);

VSCATTERPFODPS void \_\_mm512\_mask\_prefetch\_i32scatter\_ps(void \*base, \_\_mmask16 m, \_\_m512i vdx, int scale, int hint);

VSCATTERPFOQPD void \_\_mm512\_prefetch\_i64scatter\_pd(void \*base, \_\_m512i vdx, int scale, int hint);

VSCATTERPFOQPD void \_\_mm512\_mask\_prefetch\_i64scatter\_pd(void \*base, \_\_mmask8 m, \_\_m512i vdx, int scale, int hint);

VSCATTERPFOQPS void \_\_mm512\_prefetch\_i64scatter\_ps(void \*base, \_\_m512i vdx, int scale, int hint);

VSCATTERPFOQPS void \_\_mm512\_mask\_prefetch\_i64scatter\_ps(void \*base, \_\_mmask8 m, \_\_m512i vdx, int scale, int hint);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.

## VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /6 /vsib VSCATTERPF1DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /6 /vsib VSCATTERPF1QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /6 /vsib VSCATTERPF1DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /6 /vsib VSCATTERPF1QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

### Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

- (1) The prefetches may happen in any order (or not at all). The instruction is a hint.
- (2) The mask is left unchanged.
- (3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- (4) No FP nor memory faults may be produced by this instruction.
- (5) Prefetches do not handle cache line splits
- (6) A #UD is signaled if the memory operand is encoded without the SIB byte.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a vector register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

**VSCATTERPF1DPS (EVEX encoded version)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1

i ← j \* 32

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[j+31:i]) \* SCALE + DISP], Level=1, RFO = 1)

FI;

ENDFOR

**VSCATTERPF1DPD (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 32

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP], Level=1, RFO = 1)

FI;

ENDFOR

**VSCATTERPF1QPS (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[j+63:i]) \* SCALE + DISP], Level=1, RFO = 1)

FI;

ENDFOR

**VSCATTERPF1QPD (EVEX encoded version)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1

i ← j \* 64

k ← j \* 64

IF k1[j]

Prefetch( [BASE\_ADDR + SignExtend(VINDEX[k+63:k]) \* SCALE + DISP], Level=1, RFO = 1)

FI;

ENDFOR

**Intel C/C++ Compiler Intrinsic Equivalent**

VSCATTERPF1DPD void \_\_mm512\_prefetch\_i32scatter\_pd(void \*base, \_\_m256i vdx, int scale, int hint);

VSCATTERPF1DPD void \_\_mm512\_mask\_prefetch\_i32scatter\_pd(void \*base, \_\_mmask8 m, \_\_m256i vdx, int scale, int hint);

VSCATTERPF1DPS void \_\_mm512\_prefetch\_i32scatter\_ps(void \*base, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1DPS void \_\_mm512\_mask\_prefetch\_i32scatter\_ps(void \*base, \_\_mmask16 m, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1QPD void \_\_mm512\_prefetch\_i64scatter\_pd(void \*base, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1QPD void \_\_mm512\_mask\_prefetch\_i64scatter\_pd(void \*base, \_\_mmask8 m, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1QPS void \_\_mm512\_prefetch\_i64scatter\_ps(void \*base, \_\_m512i vdx, int scale, int hint);

VSCATTERPF1QPS void \_\_mm512\_mask\_prefetch\_i64scatter\_ps(void \*base, \_\_mmask8 m, \_\_m512i vdx, int scale, int hint);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.