CLDEMOTE—Cache Line Demote

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 1C /0 CLDEMOTE m8	A	V/V	CLDEMOTE	Hint to hardware to move the cache line containing m8 to a more distant level of the cache without writing back to memory.

Instruction Operand Encoding[1]

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

Description

Hints to hardware that the cache line that contains the linear address specified with the memory operand should be moved (“demoted”) from the cache(s) closest to the processor core to a level more distant from the processor core. This may accelerate subsequent accesses to the line by other cores in the same coherence domain, especially if the line was written by the core that demotes the line. Moving the line in such a manner is a performance optimization, i.e., it is a hint which does not modify architectural state. Hardware may choose which level in the cache hierarchy to retain the line (e.g., L3 in typical server designs). The source operand is a byte memory location.

The availability of the CLDEMOTE instruction is indicated by the presence of the CPUID feature flag CLDEMOTE (bit 25 of the ECX register in sub-leaf 07H, see “CPUID—CPU Identification” in Chapter 1). On processors which do not support the CLDEMOTE instruction (including legacy hardware) the instruction will be treated as a NOP.

A CLDEMOTE instruction is ordered with respect to stores to the same cache line, but unordered with respect to other instructions including memory fences, CLDEMOTE, CLWB or CLFLUSHOPT instructions to a different cache line. Since CLDEMOTE will retire in order with respect to stores to the same cache line, software should ensure that after issuing CLDEMOTE the line is not accessed again immediately by the same core to avoid cache data movement penalties.

The effective memory type of the page containing the affected line determines the effect; cacheable types are likely to generate a data movement operation, while uncacheable types may cause the instruction to be ignored.

Speculative fetching can occur at any time and is not tied to instruction execution. The CLDEMOTE 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 CLDEMOTE instruction that references the cache line.

Unlike CLFLUSH, CLFLUSHOPT and CLWB instructions, CLDEMOTE is not guaranteed to write back modified data to memory.

The CLDEMOTE instruction may be ignored by hardware in certain cases and is not a guarantee.

The CLDEMOTE instruction can be used at all privilege levels. In certain processor implementations the CLDEMOTE instruction may set the A bit but not the D bit in the page tables.

If the line is not found in the cache, the instruction will be treated as a NOP.

In some implementations, the CLDEMOTE instruction may always cause a transactional abort with Transactional Synchronization Extensions (TSX). However, programmers must not rely on CLDEMOTE instruction to force a transactional abort.

1. ModRM.MOD != 011B

Operation

```
Cache_Line_Demote(m8);
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

CLDEMOTE void _cldemote(const 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

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

GF2P8AFFINEINVQB — Galois Field Affine Transformation Inverse

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CF /r /ib GF2P8AFFINEINVQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.NDS.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
VEX.NDS.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.NDS.128.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.NDS.256.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes inverse affine transformation in the finite field GF(2^8).
EVEX.NDS.512.66.0F3A.W1 CF /r /ib VGF2P8AFFINEINVQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F GFNI	Computes inverse affine transformation in the finite field GF(2^8).

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	NA
B	NA	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEINVB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by A * inv(x) + b where “A” is an 8 by 8 bit matrix, and “x” and “b” are 8-bit vectors. The inverse of the bytes in x is defined with respect to the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

One SIMD register (operand 1) holds “x” as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 “A” values, which are operated upon by the correspondingly aligned 8 “x” values in the first register. The “b” vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require 16B alignment on their memory operations.

The inverse of each byte is given by the following table. The upper nibble is on the vertical axis and the lower nibble is on the horizontal axis. For example, the inverse of 0x95 is 0x8A.

Table 2-1. Inverse Byte Listings

-	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
2	3A	6E	5A	F1	55	4D	A8	C9	C1	A	98	15	30	44	A2	C2
3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	9
5	ED	5C	5	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	6	A1	FA	81	82
8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	2	B9	A4
9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
B	C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
C	B	28	2F	A3	DA	D4	E4	F	A9	27	53	4	1B	FC	AC	E6
D	7A	7	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
E	B1	D	D6	EB	C6	E	CF	AD	8	4E	D7	E3	5D	50	1E	B3
F	5B	23	38	34	68	46	3	8C	DD	9C	7D	A0	CD	1A	41	1C

Operation

```
define affine_inverse_byte(tsrc2qw, src1byte, imm):
    FOR i ← 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        * inverse(x) is defined in the table above *
        retbyte.bit[i] ← parity(tsrc2qw.byte[7-i] AND inverse(src1byte)) XOR imm8.bit[i]
    return retbyte
```

VGF2P8AFFINEINVQB dest, src1, src2, imm8 (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 ← SRC2.qword[0]
    ELSE:
        tsrc2 ← SRC2.qword[j]

FOR b ← 0 to 7:
    IF k1[j*8+b] OR *no writemask*:
        FOR i ← 0 to 7:
            DEST.qword[j].byte[b] ← affine_inverse_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
    ELSE IF *zeroing*:
        DEST.qword[j].byte[b] ← 0
    *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VGF2P8AFFINEINVQB dest, src1, src2, imm8 (128b and 256b VEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256)
FOR j ← 0 TO KL-1:
    FOR b ← 0 to 7:
        DEST.qword[j].byte[b] ← affine_inverse_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] ← 0
```

GF2P8AFFINEINVQB srcdest, src1, imm8 (128b SSE encoded version)

```
FOR j ← 0 TO 1:
    FOR b ← 0 to 7:
        SRCDEST.qword[j].byte[b] ← affine_inverse_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

Intel C/C++ Compiler Intrinsic Equivalent

GF2P8AFFINEINVQB __m128i _mm_gf2p8affineinv_epi64_epi8(__m128i, __m128i, int);
GF2P8AFFINEINVQB __m128i _mm_mask_gf2p8affineinv_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
GF2P8AFFINEINVQB __m128i _mm_maskz_gf2p8affineinv_epi64_epi8(__mmask16, __m128i, __m128i, int);
GF2P8AFFINEINVQB __m256i _mm256_gf2p8affineinv_epi64_epi8(__m256i, __m256i, int);
GF2P8AFFINEINVQB __m256i _mm256_mask_gf2p8affineinv_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
GF2P8AFFINEINVQB __m256i _mm256_maskz_gf2p8affineinv_epi64_epi8(__mmask32, __m256i, __m256i, int);
GF2P8AFFINEINVQB __m512i _mm512_gf2p8affineinv_epi64_epi8(__m512i, __m512i, int);
GF2P8AFFINEINVQB __m512i _mm512_mask_gf2p8affineinv_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
GF2P8AFFINEINVQB __m512i _mm512_maskz_gf2p8affineinv_epi64_epi8(__mmask64, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

GF2P8AFFINEQB — Galois Field Affine Transformation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F3A CE /r /ib GF2P8AFFINEQB xmm1, xmm2/m128, imm8	A	V/V	GFNI	Computes affine transformation in the finite field GF(2^8).
VEX.NDS.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field GF(2^8).
VEX.NDS.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.NDS.128.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.NDS.256.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL GFNI	Computes affine transformation in the finite field GF(2^8).
EVEX.NDS.512.66.0F3A.W1 CE /r /ib VGF2P8AFFINEQB zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F GFNI	Computes affine transformation in the finite field GF(2^8).

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	imm8 (r)	NA
B	NA	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

Description

The AFFINEB instruction computes an affine transformation in the Galois Field 2^8. For this instruction, an affine transformation is defined by A * x + b where “A” is an 8 by 8 bit matrix, and “x” and “b” are 8-bit vectors. One SIMD register (operand 1) holds “x” as either 16, 32 or 64 8-bit vectors. A second SIMD (operand 2) register or memory operand contains 2, 4, or 8 “A” values, which are operated upon by the correspondingly aligned 8 “x” values in the first register. The “b” vector is constant for all calculations and contained in the immediate byte.

The EVEX encoded form of this instruction does not support memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

Operation

```
define parity(x):
    t ← 0                    // single bit
    FOR i ← 0 to 7:
        t = t xor x.bit[i]
    return t

define affine_byte(tsrc2qw, src1byte, imm):
    FOR i ← 0 to 7:
        * parity(x) = 1 if x has an odd number of 1s in it, and 0 otherwise.*
        retbyte.bit[i] ← parity(tsrc2qw.byte[7-i] AND src1byte) XOR imm8.bit[i]
    return retbyte
```

VGF2P8AFFINEQB dest, src1, src2, imm8 (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC2 is memory and EVEX.b==1:
        tsrc2 ← SRC2.qword[0]
    ELSE:
        tsrc2 ← SRC2.qword[j]

    FOR b ← 0 to 7:
        IF k1[j*8+b] OR *no writemask*:
            DEST.qword[j].byte[b] ← affine_byte(tsrc2, SRC1.qword[j].byte[b], imm8)
        ELSE IF *zeroing*:
            DEST.qword[j].byte[b] ← 0
        *ELSE DEST.qword[j].byte[b] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VGF2P8AFFINEQB dest, src1, src2, imm8 (128b and 256b VEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256)
FOR j ← 0 TO KL-1:
    FOR b ← 0 to 7:
        DEST.qword[j].byte[b] ← affine_byte(SRC2.qword[j], SRC1.qword[j].byte[b], imm8)
DEST[MAX_VL-1:VL] ← 0
```

GF2P8AFFINEQB srcdest, src1, imm8 (128b SSE encoded version)

```
FOR j ← 0 TO 1:
    FOR b ← 0 to 7:
        SRCDEST.qword[j].byte[b] ← affine_byte(SRC1.qword[j], SRCDEST.qword[j].byte[b], imm8)
```

Intel C/C++ Compiler Intrinsic Equivalent

GF2P8AFFINEQB __m128i _mm_gf2p8affine_epi64_epi8(__m128i, __m128i, int);
GF2P8AFFINEQB __m128i _mm_mask_gf2p8affine_epi64_epi8(__m128i, __mmask16, __m128i, __m128i, int);
GF2P8AFFINEQB __m128i _mm_maskz_gf2p8affine_epi64_epi8(__mmask16, __m128i, __m128i, int);
GF2P8AFFINEQB __m256i _mm256_gf2p8affine_epi64_epi8(__m256i, __m256i, int);
GF2P8AFFINEQB __m256i _mm256_mask_gf2p8affine_epi64_epi8(__m256i, __mmask32, __m256i, __m256i, int);
GF2P8AFFINEQB __m256i _mm256_maskz_gf2p8affine_epi64_epi8(__mmask32, __m256i, __m256i, int);
GF2P8AFFINEQB __m512i _mm512_gf2p8affine_epi64_epi8(__m512i, __m512i, int);
GF2P8AFFINEQB __m512i _mm512_mask_gf2p8affine_epi64_epi8(__m512i, __mmask64, __m512i, __m512i, int);
GF2P8AFFINEQB __m512i _mm512_maskz_gf2p8affine_epi64_epi8(__mmask64, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: Exceptions Type 4.
EVEX-encoded: See Exceptions Type E4NF.

GF2P8MULB — Galois Field Multiply Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F38 CF /r GF2P8MULB xmm1, xmm2/m128	A	V/V	GFNI	Multiplies elements in the finite field GF(2^8).
VEX.NDS.128.66.0F38.W0 CF /r VGF2P8MULB xmm1, xmm2, xmm3/m128	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
VEX.NDS.256.66.0F38.W0 CF /r VGF2P8MULB ymm1, ymm2, ymm3/m256	B	V/V	AVX GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.NDS.128.66.0F38.W0 CF /r VGF2P8MULB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.NDS.256.66.0F38.W0 CF /r VGF2P8MULB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL GFNI	Multiplies elements in the finite field GF(2^8).
EVEX.NDS.512.66.0F38.W0 CF /r VGF2P8MULB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512F GFNI	Multiplies elements in the finite field GF(2^8).

Instruction Operand Encoding

Op/En	Tuple	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)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

The instruction multiplies elements in the finite field GF(2^8), operating on a byte (field element) in the first source operand and the corresponding byte in a second source operand. The field GF(2^8) is represented in polynomial representation with the reduction polynomial $x^8 + x^4 + x^3 + x + 1$.

This instruction does not support broadcasting.

The EVEX encoded form of this instruction supports memory fault suppression. The SSE encoded forms of the instruction require16B alignment on their memory operations.

Operation

```
define gf2p8mul_byte(src1byte, src2byte):
    tword ← 0
    FOR i ← 0 to 7:
        IF src2byte.bit[i]:
            tword ← tword XOR (src1byte<< i)
        * carry out polynomial reduction by the characteristic polynomial p*
    FOR i ← 14 downto 8:
        p ← 0x11B << (i-8)          *0x11B = 0000_0001_0001_1011 in binary*
        IF tword.bit[i]:
            tword ← tword XOR p
return tword.byte[0]
```

VGF2P8MULB dest, src1, src2 (EVEX encoded version)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[j] ← gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
    ELSE iF *zeroing*:
        DEST.byte[j] ← 0
    * ELSE DEST.byte[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VGF2P8MULB dest, src1, src2 (128b and 256b VEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256)
FOR j ← 0 TO KL-1:
    DEST.byte[j] ← gf2p8mul_byte(SRC1.byte[j], SRC2.byte[j])
DEST[MAX_VL-1:VL] ← 0
```

GF2P8MULB srcdest, src1 (128b SSE encoded version)

```
FOR j ← 0 TO 15:
    SRCDEST.byte[j] ← gf2p8mul_byte(SRCDEST.byte[j], SRC1.byte[j])
```

Intel C/C++ Compiler Intrinsic Equivalent

VGF2P8MULB __m128i _mm_gf2p8mul_epi8(__m128i, __m128i);
VGF2P8MULB __m128i _mm_mask_gf2p8mul_epi8(__m128i, __mmask16, __m128i, __m128i);
VGF2P8MULB __m128i _mm_maskz_gf2p8mul_epi8(__mmask16, __m128i, __m128i);
VGF2P8MULB __m256i _mm256_gf2p8mul_epi8(__m256i, __m256i);
VGF2P8MULB __m256i _mm256_mask_gf2p8mul_epi8(__m256i, __mmask32, __m256i, __m256i);
VGF2P8MULB __m256i _mm256_maskz_gf2p8mul_epi8(__mmask32, __m256i, __m256i);
VGF2P8MULB __m512i _mm512_gf2p8mul_epi8(__m512i, __m512i);
VGF2P8MULB __m512i _mm512_mask_gf2p8mul_epi8(__m512i, __mmask64, __m512i, __m512i);
VGF2P8MULB __m512i _mm512_maskz_gf2p8mul_epi8(__mmask64, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

Legacy-encoded and VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

MOVDIRI—Move Doubleword as Direct Store

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 F9 /r MOVDIRI m32, r32	A	V/V	MOVDIRI	Move doubleword from r32 to m32 using direct store.
NP REX.W + 0F 38 F9 /r MOVDIRI m64, r64	A	V/N.E.	MOVDIRI	Move quadword from r64 to m64 using direct store.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	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 direct-store operation. The source operand is a general purpose register. The destination operand is a 32-bit memory location (MODRM.MOD != 0b11). 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 summary chart at the beginning of this section for encoding data and limits.

The direct-store is implemented by using write combining (WC) memory type protocol for writing data. 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. If the destination address is cached, the line is written-back (if modified) and invalidated from the cache, before the direct-store. Unlike stores with non-temporal hint that allow uncached (UC) and write-protected (WP) memory-type for the destination to override the non-temporal hint, direct-stores always follow WC memory type protocol irrespective of the destination address memory type (including UC and WP types).

Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combing buffer may be combined with younger direct stores to the same address. Because WC protocol used by direct-stores follows a weakly-ordered memory consistency model, a fencing operation using SFENCE or MFENCE should follow the MOVDIRI instruction to enforce ordering when needed.

Direct-stores issued by MOVDIRI to a destination aligned to a 4-byte boundary (8-byte boundary if used with REX.W prefix) guarantee 4-byte (8-byte with REX.W prefix) write-completion atomicity. This means that the data arrives at the destination in a single undivided 4-byte (or 8-byte) write transaction. If the destination is not aligned for the write size, the direct-stores issued by MOVDIRI are split and arrive at the destination in two parts. Each part of such split direct-store will not merge with younger stores but can arrive at the destination in either order. Availability of the MOVDIRI instruction is indicated by the presence of the CPUID feature flag MOVDIRI (bit 27 of the ECX register in leaf 07H, see "CPUID — CPU Identification" in Chapter 1).

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVDIRI void _directstoreu_u32(void *dst, uint32_t val)
MOVDIRI void _directstoreu_u64(void *dst, uint64_t val)

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.07H.0H:ECX.MOVDIRI[bit 27] = 0.
	If LOCK prefix or operand-size (66H) prefix is used.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.07H.0H:ECX.MOVDIRI[bit 27] = 0.
	If LOCK prefix or operand-size (66H) prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF (fault-code)	For a page fault.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.MOVDIRI[bit 27] = 0.
	If LOCK prefix or operand-size (66H) prefix is used.
#AC	If alignment checking is enabled and an unaligned memory reference made while in current privilege level 3.

MOVDIR64B—Move 64 Bytes as Direct Store

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F8 /r MOVDIR64B r16/r32/r64, m512	A	V/V	MOVDIR64B	Move 64-bytes as direct-store with guaranteed 64-byte write atomicity from the source memory operand address to destination memory address specified as offset to ES segment in the register operand.

Instruction Operand Encoding

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

Description

Moves 64-bytes as direct-store with 64-byte write atomicity from source memory address to destination memory address. The source operand is a normal memory operand (MODRM.MOD != 0b11). The destination operand is a memory location specified in a general-purpose register. The register content is interpreted as an offset into ES segment without any segment override. In 64-bit mode, the register operand width is 64-bits (32-bits with 67H prefix). Outside of 64-bit mode, the register width is 32-bits when CS.D=1 (16-bits with 67H prefix), and 16-bits when CS.D=0 (32-bits with 67H prefix). MOVDIR64B requires the destination address to be 64-byte aligned. No alignment restriction is enforced for source operand.

MOVDIR64B reads 64-bytes from the source memory address and performs a 64-byte direct-store operation to the destination address. The load operation follows normal read ordering based on source address memory-type. The direct-store is implemented by using the write combining (WC) memory type protocol for writing data. 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. If the destination address is cached, the line is written-back (if modified) and invalidated from the cache, before the direct-store.

Unlike stores with non-temporal hint which allow UC/WP memory-type for destination to override the non-temporal hint, direct-stores always follow WC memory type protocol irrespective of destination address memory type (including UC/WP types). Unlike WC stores and stores with non-temporal hint, direct-stores are eligible for immediate eviction from the write-combining buffer, and thus not combined with younger stores (including direct-stores) to the same address. Older WC and non-temporal stores held in the write-combing buffer may be combined with younger direct stores to the same address. Because WC protocol used by direct-stores follow weakly-ordered memory consistency model, fencing operation using SFENCE or MFENCE should follow the MOVDIR64B instruction to enforce ordering when needed.

There is no atomicity guarantee provided for the 64-byte load operation from source address, and processor implementations may use multiple load operations to read the 64-bytes. The 64-byte direct-store issued by MOVDIR64B guarantees 64-byte write-completion atomicity. This means that the data arrives at the destination in a single undivided 64-byte write transaction.

Availability of the MOVDIR64B instruction is indicated by the presence of the CPUID feature flag MOVDIR64B (bit 28 of the ECX register in leaf 07H, see “CPUID — CPU Identification” in Chapter 1).

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVDIR64B void _movdir64b(void *dst, const void* src)

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If address in destination (register) operand is not aligned to a 64-byte boundary.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0.
	If 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.
	If address in destination (register) operand is not aligned to a 64-byte boundary.
#UD	If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0.
	If 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 memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
	If address in destination (register) operand is not aligned to a 64-byte boundary.
#PF (fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.MOVDIR64B[bit 28] = 0.
	If LOCK prefix is used.

PCONFIG — Platform Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C5 PCONFIG	A	V/V	PCONFIG	This instruction is used to execute functions for configuring platform features. EAX: Leaf function to be invoked. RBX/RCX/RDX: Leaf-specific purpose.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA	NA

Description

PCONFIG allows software to configure certain platform features. PCONFIG supports multiple leaf functions, with a leaf function identified by the value in EAX. The registers RBX, RCX, and RDX have leaf-specific purposes.

Each PCONFIG leaf function applies to a specific hardware block called a PCONFIG target, and each PCONFIG target is associated with a numerical identifier. The identifiers of the PCONFIG targets supported by the CPU (which imply the supported leaf functions) are enumerated in the sub-leaves of the PCONFIG-information leaf of CPUID (EAX = 1BH). An attempt to execute an undefined leaf function results in a general-protection exception (#GP).

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 || CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 && CS.L = 1). The value of CS.D has no effect on address calculation.

Table 2-2 shows the leaf encodings for PCONFIG.

Table 2-2. PCONFIG Leaf Encodings

Leaf	Encoding	Description
MKTME_KEY_PROGRAM	00000000H	This leaf is used to program the key and encryption mode associated with a KeyID.
RESERVED	00000001H - FFFFFFFFH	Reserved for future use (#GP(0) if used).

The MKTME_KEY_PROGRAM leaf of PCONFIG pertains to the MKTME target, which has target identifier 1. It is used by software to manage the key associated with a KeyID. The leaf function is invoked by setting the leaf value of 0 in EAX and the address of MKTME_KEY_PROGRAM_STRUCT in RBX. Successful execution of the leaf clears RAX (set to zero) and ZF, CF, PF, AF, OF, and SF are cleared. In case of failure, the failure reason is indicated in RAX with ZF set to 1 and CF, PF, AF, OF, and SF are cleared. The MKTME_KEY_PROGRAM leaf uses the MKTME_KEY_PROGRAM_STRUCT in memory shown in Table 2-3.

Table 2-3. MKTME_KEY_PROGRAM_STRUCT Format

Field	Offset (bytes)	Size (bytes)	Comments
KEYID	0	2	Key Identifier.
KEYID_CTRL	2	4	KeyID control: ▪ Bits [7:0]: COMMAND. ▪ Bits [23:8]: ENC_ALG. ▪ Bits [31:24]: Reserved, must be zero.
RESERVED	6	58	Reserved, must be zero.
KEY_FIELD_1	64	64	Software supplied KeyID data key or entropy for KeyID data key.
KEY_FIELD_2	128	64	Software supplied KeyID tweak key or entropy for KeyID tweak key.

A description of each of the fields in MKTME_KEY_PROGRAM_STRUCT is provided below:

- **KEYID:** Key Identifier being programmed to the MKTME engine.
- **KEYID_CTRL:** The KEYID_CTRL field carries two sub-fields used by software to control the behavior of a KeyID: Command and KeyID encryption algorithm.

  The command used controls the encryption mode for a KeyID. Table 2-4 provides a summary of the commands supported.

Table 2-4. Supported Key Programming Commands

Command	Encoding	Description
KEYID_SET_KEY_DIRECT	0	Software uses this mode to directly program a key for use with KeyID.
KEYID_SET_KEY_RANDOM	1	CPU generates and assigns an ephemeral key for use with a KeyID. Each time the instruction is executed, the CPU generates a new key using a hardware random number generator and the keys are discarded on reset.
KEYID_CLEAR_KEY	2	Clear the (software programmed) key associated with the KeyID. On execution of this command, the KeyID gets TME behavior (encrypt with platform TME key).
KEYID_NO_ENCRYPT	3	Do not encrypt memory when this KeyID is in use.

  The encryption algorithm field (ENC_ALG) allows software to select one of the activated encryption algorithms for the KeyID. The BIOS can activate a set of algorithms to allow for use when programming keys using the IA32_TME_ACTIVATE MSR (does not apply to KeyID 0 which uses TME policy). The ISA checks to ensure that the algorithm selected by software is one of the algorithms that has been activated by the BIOS.

- **KEY_FIELD_1:** This field carries the software supplied data key to be used for the KeyID if the direct key programming option is used (KEYID_SET_KEY_DIRECT). When the random key programming option is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random data key. It is software's responsibility to ensure that the key supplied for the direct programming option or the entropy supplied for the random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.
- **KEY_FIELD_2:** This field carries the software supplied tweak key to be used for the KeyID if the direct key programming option is used (KEYID_SET_KEY_DIRECT). When the random key programming option is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random tweak key. It is software's responsibility to ensure that the key supplied for the direct programming option or the entropy supplied for the random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.

  All KeyIDs use the TME key on MKTME activation. Software can at any point decide to change the key for a KeyID using the PCONFIG instruction. Change of keys for a KeyID does NOT change the state of the TLB caches or memory pipeline. It is software's responsibility to take appropriate actions to ensure correct behavior.

  Table 2-5 shows the return values associated with the MKTME_KEY_PROGRAM leaf of PCONFIG. On instruction execution, RAX is populated with the return value.

Table 2-5. Supported Key Programming Commands

Return Value	Encoding	Description
PROG_SUCCESS	0	KeyID was successfully programmed.
INVALID_PROG_CMD	1	Invalid KeyID programming command.
ENTROPY_ERROR	2	Insufficient entropy.
INVALID_KEYID	3	KeyID not valid.
INVALID_ENC_ALG	4	Invalid encryption algorithm chosen (not supported).
DEVICE_BUSY	5	Failure to access key table.

PCONFIG Virtualization

Software in VMX root mode can control the execution of PCONFIG in VMX non-root mode using the following execution controls introduced for PCONFIG:

- PCONFIG_ENABLE: This control is a single bit control and enables the PCONFIG instruction in VMX non-root mode. If 0, the execution of PCONFIG in VMX non-root mode causes #UD. Otherwise, execution of PCONFIG works according to PCONFIG_EXITING.
- PCONFIG_EXITING: This is a 64b control and allows VMX root mode to cause a VM-exit for various leaf functions of PCONFIG. This control does not have any effect if the PCONFIG_ENABLE control is clear.

PCONFIG Concurrency

In a scenario, where the MKTME_KEY_PROGRAM leaf of PCONFIG is executed concurrently on multiple logical processors, only one logical processor will succeed in updating the key table. PCONFIG execution will return with an error code (DEVICE_BUSY) on other logical processors and software must retry. In cases where the instruction execution fails with a DEVICE_BUSY error code, the key table is not updated, thereby ensuring that either the key table is updated in its entirety with the information for a KeyID, or it is not updated at all. In order to accomplish this, the MKTME_KEY_PROGRAM leaf of PCONFIG maintains a writer lock for updating the key table. This lock is referred to as the Key table lock and denoted in the instruction flows as KEY_TABLE_LOCK. The lock can either be unlocked, when no logical processor is holding the lock (also the initial state of the lock) or be in an exclusive state where a logical processor is trying to update the key table. There can be only one logical processor holding the lock in exclusive state. The lock, being exclusive, can only be acquired when the lock is in unlocked state.

PCONFIG uses the following syntax to acquire KEY_TABLE_LOCK in exclusive mode and release the lock:

- KEY_TABLE_LOCK.ACQUIRE(WRITE)
- KEY_TABLE_LOCK.RELEASE()

Operation

Table 2-6. PCONFIG Operation Variables

Variable Name	Type	Size (Bytes)	Description
TMP_KEY_PROGRAM_STRUCT	MKTME_KEY_PROGRAM_STRUCT	192	Structure holding the key programming structure.
TMP_RND_DATA_KEY	UINT128	16	Random data key generated for random key programming option.
TMP_RND_TWEAK_KEY	UINT128	16	Random tweak key generated for random key programming option.

```
(* #UD if PCONFIG is not enumerated or CPL>0 *)
if (CPUID.7.0:EDX[18] == 0 OR CPL > 0) #UD;

if (in VMX non-root mode)
{
    if (VMCS.PCONFIG_ENABLE == 1)
    {
        if ((EAX > 62 AND VMCS.PCONFIG_EXITING[63] ==1) OR
            (EAX < 63 AND VMCS.PCONFIG_EXITING[EAX] == 1))
        {
            Set VMCS.EXIT_REASON = PCONFIG; //No Exit qualification
            Deliver VMEXIT;
        }
    }
    else
    {
        #UD
```

```
    }
}

(* #GP(0) for an unsupported leaf *)
if(EAX != 0) #GP(0)

(* KEY_PROGRAM leaf flow *)
if (EAX == 0)
{
    (* #GP(0) if TME_ACTIVATE MSR is not locked or does not enable TME or multiple keys are not enabled *)
    if (IA32_TME_ACTIVATE.LOCK != 1 OR IA32_TME_ACTIVATE.ENABLE != 1 OR IA32_TME_ACTIVATE.MK_TME_KEYID_BITS == 0)
#GP(0)

    (* Check MKTME_KEY_PROGRAM_STRUCT is 256B aligned *)
    if(DS:RBX is not 256B aligned) #GP(0);

    (* Check that MKTME_KEY_PROGRAM_STRUCT is read accessible *)
    <<DS: RBX should be read accessible>>

    (* Copy MKTME_KEY_PROGRAM_STRUCT to a temporary variable *)
    TMP_KEY_PROGRAM_STRUCT = DS:RBX.*;

    (* RSVD field check *)
    if(TMP_KEY_PROGRAM_STRUCT.RSVD != 0) #GP(0);

    if(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.RSVD !=0) #GP(0);

    if(TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[63:16] != 0) #GP(0);

    if(TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[63:16] != 0) #GP(0);

    (* Check for a valid command *)
    if(TMP_KEY_PROGRAM_STRUCT. KEYID_CTRL.COMMAND is not a valid command)
    {
        RFLAGS.ZF = 1;
        RAX = INVALID_PROG_CMD;
        goto EXIT;
    }
    (* Check that the KEYID being operated upon is a valid KEYID *)
    if(TMP_KEY_PROGRAM_STRUCT.KEYID >
                2^IA32_TME_ACTIVATE.MK_TME_KEYID_BITS - 1
        OR TMP_KEY_PROGRAM_STRUCT.KEYID >
                IA32_TME_CAPABILITY.MK_TME_MAX_KEYS
        OR TMP_KEY_PROGRAM_STRUCT.KEYID == 0)
    {
        RFLAGS.ZF = 1;
        RAX = INVALID_KEYID;
        goto EXIT;
    }

    (* Check that only one algorithm is requested for the KeyID and it is one of the activated algorithms *)
    if(NUM_BITS(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG) != 1 ||
        (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG &
            IA32_TME_ACTIVATE. MK_TME_CRYPTO_ALGS == 0))
```

```
{
        RFLAGS.ZF = 1;
        RAX = INVALID_ENC_ALG;
        goto EXIT;
}
(* Try to acquire exclusive lock *)
if (NOT KEY_TABLE_LOCK.ACQUIRE(WRITE))
{
        //PCONFIG failure
        RFLAGS.ZF = 1;
        RAX = DEVICE_BUSY;
        goto EXIT;
}

(* Lock is acquired and key table will be updated as per the command
        Before this point no changes to the key table are made *)

switch(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.COMMAND)
{
case KEYID_SET_KEY_DIRECT:
        <<Write
                DATA_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1,
                TWEAK_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2,
                ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
                to MKTME Key table at index TMP_KEY_PROGRAM_STRUCT.KEYID
        >>
        break;

case KEYID_SET_KEY_RANDOM:
        TMP_RND_DATA_KEY = <<Generate a random key using hardware RNG>>
        if (NOT ENOUGH ENTROPY)
        {
                RFLAGS.ZF = 1;
                RAX = ENTROPY_ERROR;
                goto EXIT;
        }
        TMP_RND_TWEAK_KEY = <<Generate a random key using hardware RNG>>
        if (NOT ENOUGH ENTROPY)
        {
                RFLAGS.ZF = 1;
                RAX = ENTROPY_ERROR;
                goto EXIT;
        }
        (* Mix user supplied entropy to the data key and tweak key *)
        TMP_RND_DATA_KEY = TMP_RND_KEY XOR
                TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[15:0];
        TMP_RND_TWEAK_KEY = TMP_RND_TWEAK_KEY XOR
                TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[15:0];

        <<Write
                DATA_KEY=TMP_RND_DATA_KEY,
                TWEAK_KEY=TMP_RND_TWEAK_KEY,
                ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
                to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
```

```
        >>
        break;

    case KEYID_CLEAR_KEY:
        <<Write
        DATA_KEY='0,
        TWEAK_KEY='0,
        ENCRYPTION_MODE = ENCRYPT_WITH_TME_KEY,
        to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
        >>

        break;
    case KD_NO_ENCRYPT:
        <<Write
        ENCRYPTION_MODE=NO_ENCRYPTION,
        to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
        >>
        break;
    }

    RAX = 0;
    RFLAGS.ZF = 0;

    //Release Lock
    KEY_TABLE_LOCK(RELEASE);

    EXIT:
    RFLAGS.CF=0;
    RFLAGS.PF=0;
    RFLAGS.AF=0;
    RFLAGS.OF=0;
    RFLAGS.SF=0;
}

end_of_flow
```

Intel C/C++ Compiler Intrinsic Equivalent

TBD

Protected Mode Exceptions

#GP(0)	If input value in EAX encodes an unsupported leaf.
	If IA32_TME_ACTIVATE MSR is not locked.
	If TME and MKTME capability are not enabled in IA32_TME_ACTIVATE MSR.
	If the memory operand is not 256B aligned.
	If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set.
	If a memory operand effective address is outside the DS segment limit.
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	If any of the LOCK/REP/OSIZE/VEX prefixes are used.
	If current privilege level is not 0.
	If CPUID.7.0:EDX[bit 18] = 0
	If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.

Real Address Mode Exceptions

#GP	If input value in EAX encodes an unsupported leaf.
	If IA32_TME_ACTIVATE MSR is not locked.
	If TME and MKTME capability is not enabled in IA32_TME_ACTIVATE MSR.
	If a memory operand is not 256B aligned.
	If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set.
#UD	If any of the LOCK/REP/OSIZE/VEX prefixes are used.
	If current privilege level is not 0.
	If CPUID.7.0:EDX.PCONFIG[bit 18] = 0
	If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.

Virtual 8086 Mode Exceptions

#UD	PCONFIG instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If input value in EAX encodes an unsupported leaf.
	If IA32_TME_ACTIVATE MSR is not locked.
	If TME and MKTME capability is not enabled in IA32_TME_ACTIVATE MSR.
	If a memory operand is not 256B aligned.
	If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set.
	If a memory operand is non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	If any of the LOCK/REP/OSIZE/VEX prefixes are used.
	If the current privilege level is not 0.
	If CPUID.7.0:EDX.PCONFIG[bit 18] = 0.
	If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.

TPAUSE—Timed PAUSE

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F AE /6 TPAUSE r32, <edx>, <eax>	A	V/V	WAITPKG	Directs the processor to enter an implementation-dependent optimized state until the TSC reaches the value in EDX:EAX.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r)	NA	NA	NA

Description

TPAUSE instructs the processor to enter an implementation-dependent optimized state. There are two such optimized states to choose from: light-weight power/performance optimized state, and improved power/performance optimized state. The selection between the two is governed by the explicit input register bit[0] source operand.

TPAUSE is available when CPUID.7.0:ECX.WAITPKG[bit 5] is enumerated as 1. TPAUSE may be executed at any privilege level. This instruction's operation is the same in non-64-bit modes and in 64-bit mode.

Unlike PAUSE, the TPAUSE instruction will not cause an abort when used inside a transactional region, described in the chapter "Programming with Intel Transactional Synchronization Extensions" of the Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

The input register contains information such as the preferred optimized state the processor should enter as described in the following table. Bits other than bit 0 are reserved and will result in #GP if non-zero.

Table 2-7. TPAUSE Input Register Bit Definitions

Bit Value	State Name	Wakeup Time	Power Savings	Other Benefits
bit[0] = 0	C0.2	Slower	Larger	Improves performance of the other SMT thread(s).
bit[0] = 1	C0.1	Faster	Smaller	NA
bits[31:1]	NA	NA	NA	Reserved

The instruction execution wakes up when the time-stamp counter reaches or exceeds the implicit EDX:EAX 64-bit input value.

Prior to executing the TPAUSE instruction, an operating system may specify the maximum delay it allows the processor to suspend its operation. It can do so by writing TSC-quanta value to the following 32-bit MSR (IA32_UMWAIT_CONTROL at MSR index E1H):

- IA32_UMWAIT_CONTROL[31:2] — Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.
- IA32_UMWAIT_CONTROL[1] — Reserved.
- IA32_UMWAIT_CONTROL[0] — C0.2 is not allowed by the OS. Value of "1" means all C0.2 requests revert to C0.1.

If the processor that executed a TPAUSE instruction wakes due to the expiration of the operating system time-limit, the instructions sets RFLAGS.CF; otherwise, that flag is cleared.

The following additional events cause the processor to exit the implementation-dependent optimized state: a store to the read-set range within the transactional region, 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 cause the processor to exit the implementation-dependent optimized state proceeding to the instruction following TPAUSE. In addition, an external interrupt causes the processor to exit the implementation-dependent optimized state regardless of whether maskable-interrupts are inhibited

(EFLAGS.IF =0). It should be noted that if maskable-interrupts are inhibited execution will proceed to the instruction following TPAUSE.

MODRM.MOD must be 0b11 for this instruction.

Operation

```
os_deadline ← TSC+(IA32_MWAIT_CONTROL[31:2]<<2)
instr_deadline ← UINT64(EDX:EAX)

IF os_deadline < instr_deadline:
    deadline ← os_deadline
    using_os_deadline ← 1
ELSE:
    deadline ← instr_deadline
    using_os_deadline ← 0

WHILE TSC < deadline:
    implementation_dependent_optimized_state(Source register, deadline, IA32_UMWAIT_CONTROL[0])

IF using_os_deadline AND TSC > deadline:
    RFLAGS.CF ← 1
ELSE:
    RFLAGS.CF ← 0

RFLAGS.AF,PF,SF,ZF,OF ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

TPAUSE uint8_t _tpause(uint32_t control, uint64_t counter);

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#GP(0)	If src[31:1] != 0.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.
	If MODRM.MOD != 0b11.

UMONITOR—User Level Set Up Monitor Address

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F AE /6 UMONITOR r16/r32/r64	A	V/V	WAITPKG	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 contained in r16/r32/r64.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r)	NA	NA	NA

Description

The UMONITOR instruction arms address monitoring hardware using an address specified in the source register (the address range that the monitoring hardware checks for store operations can be determined by using the CPUID monitor leaf function, EAX=05H). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by UMWAIT.

The content of the source register is an effective address. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used. The address range must use memory of the write-back type. Only write-back memory is guaranteed to 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 UMONITOR 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, UMONITOR sets the A-bit but not the D-bit in page tables.

UMONITOR and UMWAIT are available when CPUID.7.0:ECX.WAITPKG[bit 5] is enumerated as 1. UMONITOR and UMWAIT may be executed at any privilege level. Except for the width of the source register, the instruction’s operation is the same in non-64-bit modes and in 64-bit mode.

UMONITOR does not interoperate with the legacy MWAIT instruction. If UMONITOR was executed prior to executing MWAIT and following the most recent execution of the legacy MONITOR instruction, MWAIT will not enter an optimized state. Execution will continue to the instruction following MWAIT.

The UMONITOR instruction causes a transactional abort when used inside a transactional region.

The width of the source register (16b, 32b or 64b) is determined by the effective addressing width, which is affected in the standard way by the machine mode settings and 67 prefix.

Operation

UMONITOR sets up an address range for the monitor hardware using the content of source register as an effective address and puts the monitor hardware in armed state. A store to the specified address range will trigger the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

UMONITOR void _umonitor(void *address);

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0)	If the specified segment is not SS and the source register is outside the specified segment limit.
	If the specified segment register contains a NULL segment selector.
#SS(0)	If the specified segment is SS and the source register is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.
	If MODRM.MOD != 0b11.

Real Address Mode Exceptions

#GP	If the specified segment is not SS and the source register is outside of the effective address space from 0 to FFFFH.
#SS	If the specified segment is SS and the source register is outside of the effective address space from 0 to FFFFH.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.

Virtual 8086 Mode Exceptions

Same exceptions as in real address mode; additionally:

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the specified segment is not SS and the linear address is in non-canonical form.
#SS(0)	If the specified segment is SS and the source register is in non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.
	If MODRM.MOD != 0b11.

UMWAIT—User Level Monitor Wait

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F AE /6 UMWAIT r32, <edx>, <eax>	A	V/V	WAITPKG	A hint that allows 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	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r)	NA	NA	NA

Description

UMWAIT instructs the processor to enter an implementation-dependent optimized state while monitoring a range of addresses. The optimized state may be either a light-weight power/performance optimized state or an improved power/performance optimized state. The selection between the two states is governed by the explicit input register bit[0] source operand.

UMWAIT is available when CPUID.7.0:ECX.WAITPKG[bit 5] is enumerated as 1. UMWAIT may be executed at any privilege level. This instruction's operation is the same in non-64-bit modes and in 64-bit mode.

The input register contains information such as the preferred optimized state the processor should enter as described in the following table. Bits other than bit 0 are reserved and will result in #GP if nonzero.

Table 2-8. UMWAIT Input Register Bit Definitions

Bit Value	State Name	Wakeup Time	Power Savings	Other Benefits
bit[0] = 0	C0.2	Slower	Larger	Improves performance of the other SMT thread(s) on the same core.
bit[0] = 1	C0.1	Faster	Smaller	NA
bits[31:1]	NA	NA	NA	Reserved

The instruction wakes up when the time-stamp counter reaches or exceeds the implicit EDX:EAX 64-bit input value (if the monitoring hardware did not trigger beforehand).

Prior to executing the UMWAIT instruction, an operating system may specify the maximum delay it allows the processor to suspend its operation. It can do so by writing TSC-quanta value to the following 32bit MSR (IA32_UMWAIT_CONTROL at MSR index E1H):

- IA32_UMWAIT_CONTROL[31:2] — Determines the maximum time in TSC-quanta that the processor can reside in either C0.1 or C0.2. A zero value indicates no maximum time. The maximum time value is a 32-bit value where the upper 30 bits come from this field and the lower two bits are zero.
- IA32_UMWAIT_CONTROL[1] — Reserved.
- IA32_UMWAIT_CONTROL[0] — C0.2 is not allowed by the OS. Value of "1" means all C0.2 requests revert to C0.1.

If the processor that executed a UMWAIT instruction wakes due to the expiration of the operating system time-limit, the instructions sets RFLAGS.CF; otherwise, that flag is cleared.

The UMWAIT instruction causes a transactional abort when used inside a transactional region.

The UMWAIT instruction operates with the UMONITOR instruction. The two instructions allow the definition of an address at which to wait (UMONITOR) and an implementation-dependent optimized operation to perform while waiting (UMWAIT). The execution of UMWAIT 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 UMONITOR.

The following additional events cause the processor to exit the implementation-dependent optimized state: a store to the address range armed by the UMONITOR 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 regardless of whether maskable-interrupts are inhibited (EFLAGS.IF =0).

Following exit from the implementation-dependent-optimized state, control passes to the instruction after the UMWAIT 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 UMWAIT instruction does not restart at the UMWAIT instruction following the handling of an SMI.

If the preceding UMONITOR instruction did not successfully arm an address range or if UMONITOR was not executed prior to executing UMWAIT and following the most recent execution of the legacy MONITOR instruction (UMWAIT does not interoperate with MONITOR), then the processor will not enter an optimized state. Execution will continue to the instruction following UMWAIT.

A store to the address range armed by the UMONITOR instruction will cause the processor to exit UMWAIT if either the store was originated by other processor agents or the store was originated by a non-processor agent.

MODRM.MOD must be 0b11 for this instruction.

Operation

```
os_deadline ← TSC+(IA32_MWAIT_CONTROL[31:2]<<2)
instr_deadline ← UINT64(EDX:EAX)

IF os_deadline < instr_deadline:
    deadline ← os_deadline
    using_os_deadline ← 1
ELSE:
    deadline ← instr_deadline
    using_os_deadline ← 0

WHILE monitor hardware armed AND TSC < deadline:
    implementation_dependent_optimized_state(Source register, deadline, IA32_UMWAIT_CONTROL[0] )

IF using_os_deadline AND TSC > deadline:
    RFLAGS.CF ← 1
ELSE:
    RFLAGS.CF ← 0

RFLAGS.AF,PF,SF,ZF,OF ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

UMWAIT uint8_t _umwait(uint32_t control, uint64_t counter);

Numeric Exceptions

None

Exceptions (All Operating Modes)

#GP(0)	If src[31:1] != 0.
#UD	If CPUID.7.0:ECX.WAITPKG[bit 5]=0.
	If MODRM.MOD != 0b11.

VAESDEC — Perform One Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	A	V/V	VAES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from ymm3/m256; store the result in ymm1.
EVEX.NDS.128.66.0F38.WIG DE /r VAESDEC xmm1, xmm2, xmm3/m128	B	V/V	AVX512VL VAES	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.
EVEX.NDS.256.66.0F38.WIG DE /r VAESDEC ymm1, ymm2, ymm3/m256	B	V/V	AVX512VL VAES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from ymm3/m256; store the result in ymm1.
EVEX.NDS.512.66.0F38.WIG DE /r VAESDEC zmm1, zmm2, zmm3/m512	B	V/V	AVX512F VAES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from zmm2 with a 128-bit round key from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Mem	ModRM:reg (w)	EVEX.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 AESDEC-CLAST instruction.

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

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 (128b and 256b VEX encoded versions)

```
(KL,V) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← InvShiftRows( STATE )
    STATE ← InvSubBytes( STATE )
    STATE ← InvMixColumns( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[MAXVL-1:VL] ← 0
```

VAESDEC (EVEX encoded version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← InvShiftRows( STATE )
    STATE ← InvSubBytes( STATE )
    STATE ← InvMixColumns( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VAESDEC __m256i _mm256_aesdec_epi128(__m256i, __m256i);
VAESDEC __m512i _mm512_aesdec_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

VAESDECLAST — Perform Last Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	A	V/V	VAES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from ymm3/m256; store the result in ymm1.
EVEX.NDS.128.66.0F38.WIG DF /r VAESDECLAST xmm1, xmm2, xmm3/m128	B	V/V	AVX512VL VAES	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.
EVEX.NDS.256.66.0F38.WIG DF /r VAESDECLAST ymm1, ymm2, ymm3/m256	B	V/V	AVX512VL VAES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from ymm3/m256; store the result in ymm1.
EVEX.NDS.512.66.0F38.WIG DF /r VAESDECLAST zmm1, zmm2, zmm3/m512	B	V/V	AVX512F VAES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from zmm2 with a 128-bit round key from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

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

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESDECLAST

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

VAESDECLAST (128b and 256b VEX encoded versions)

```
(KL,VL) = (1,128), (2,256)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← InvShiftRows( STATE )
    STATE ← InvSubBytes( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[MAXVL-1:VL] ← 0
```

VAESDECLAST (EVEX encoded version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← InvShiftRows( STATE )
    STATE ← InvSubBytes( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VAESDECLAST __m256i _mm256_aesdeclast_epi128(__m256i, __m256i);
VAESDECLAST __m512i _mm512_aesdeclast_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

VAESENC — Perform One Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	A	V/V	VAES	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from the ymm3/m256; store the result in ymm1.
EVEX.NDS.128.66.0F38.WIG DC /r VAESENC xmm1, xmm2, xmm3/m128	B	V/V	AVX512VL VAES	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.
EVEX.NDS.256.66.0F38.WIG DC /r VAESENC ymm1, ymm2, ymm3/m256	B	V/V	AVX512VL VAES	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from ymm2 with a 128-bit round key from the ymm3/m256; store the result in ymm1.
EVEX.NDS.512.66.0F38.WIG DC /r VAESENC zmm1, zmm2, zmm3/m512	B	V/V	AVX512F VAES	Perform one round of an AES encryption flow, operating on a 128-bit data (state) from zmm2 with a 128-bit round key from the zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

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

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

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 (128b and 256b VEX encoded versions)

```
(KL,VL) = (1,128), (2,256)
FOR I ← 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← ShiftRows( STATE )
    STATE ← SubBytes( STATE )
    STATE ← MixColumns( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[MAXVL-1:VL] ← 0
```

VAESENC (EVEX encoded version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i ← 0 to KL-1:
    STATE ← SRC1.xmm[i] // xmm[i] is the i'th xmm word in the SIMD register
    RoundKey ← SRC2.xmm[i]
    STATE ← ShiftRows( STATE )
    STATE ← SubBytes( STATE )
    STATE ← MixColumns( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VAESENC __m256i _mm256_aesenc_epi128(__m256i, __m256i);
VAESENC __m512i _mm512_aesenc_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

VAESENCLAST — Perform Last Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	A	V/V	VAES	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from ymm2 with a 128 bit round key from ymm3/m256; store the result in ymm1.
EVEX.NDS.128.66.0F38.WIG DD /r VAESENCLAST xmm1, xmm2, xmm3/m128	B	V/V	AVX512VL VAES	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.
EVEX.NDS.256.66.0F38.WIG DD /r VAESENCLAST ymm1, ymm2, ymm3/m256	B	V/V	AVX512VL VAES	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from ymm2 with a 128 bit round key from ymm3/m256; store the result in ymm1.
EVEX.NDS.512.66.0F38.WIG DD /r VAESENCLAST zmm1, zmm2, zmm3/m512	B	V/V	AVX512F VAES	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from zmm2 with a 128 bit round key from zmm3/m512; store the result in zmm1.

Instruction Operand Encoding

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

VEX and EVEX encoded versions of the instruction allows 3-operand (non-destructive) operation. The legacy encoded versions of the instruction require that the first source operand and the destination operand are the same and must be an XMM register.

The EVEX encoded form of this instruction does not support memory fault suppression.

Operation

AESENCLAST

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

VAESENCLAST (128b and 256b VEX encoded versions)

```
(KL, VL) = (1,128), (2,256)
FOR I=0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← ShiftRows( STATE )
    STATE ← SubBytes( STATE )
    DEST.xmm[i]← STATE XOR RoundKey
DEST[MAXVL-1:VL] ← 0
```

VAESENCLAST (EVEX encoded version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    STATE ← SRC1.xmm[i]
    RoundKey ← SRC2.xmm[i]
    STATE ← ShiftRows( STATE )
    STATE ← SubBytes( STATE )
    DEST.xmm[i] ← STATE XOR RoundKey
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VAESENCLAST __m256i _mm256_aesenclast_epi128(__m256i, __m256i);
VAESENCLAST __m512i _mm512_aesenclast_epi128(__m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

VPCLMULQDQ — Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8	A	V/V	VPCLMULQDQ	Carry-less multiplication of one quadword of ymm2 by one quadword of ymm3/m256, stores the 128-bit result in ymm1. The immediate is used to determine which quadwords of ymm2 and ymm3/m256 should be used.
EVEX.NDS.128.66.0F3A.WIG 44 /r /ib VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX512VL VPCLMULQDQ	Carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, stores the 128-bit result in xmm1. The immediate is used to determine which quadwords of xmm2 and xmm3/m128 should be used.
EVEX.NDS.256.66.0F3A.WIG 44 /r /ib VPCLMULQDQ ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX512VL VPCLMULQDQ	Carry-less multiplication of one quadword of ymm2 by one quadword of ymm3/m256, stores the 128-bit result in ymm1. The immediate is used to determine which quadwords of ymm2 and ymm3/m256 should be used.
EVEX.NDS.512.66.0F3A.WIG 44 /r /ib VPCLMULQDQ zmm1, zmm2, zmm3/m512, imm8	B	V/V	AVX512F VPCLMULQDQ	Carry-less multiplication of one quadword of zmm2 by one quadword of zmm3/m512, stores the 128-bit result in zmm1. The immediate is used to determine which quadwords of zmm2 and zmm3/m512 should be used.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8 (r)

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 the table below, other bits of the immediate byte are ignored.

The EVEX encoded form of this instruction does not support memory fault suppression.

Table 2-9. PCLMULQDQ Quadword Selection of Immediate Byte

imm[4]	imm[0]	PCLMULQDQ Operation
0	0	CL_MUL(SRC2[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:

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 a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. Bits (VL_MAX-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simply programming and emit the required encoding for imm8.

Table 2-10. Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ xmm1, xmm2	0000_0000B
PCLMULHQLQDQ xmm1, xmm2	0000_0001B
PCLMULLQHQDQ xmm1, xmm2	0001_0000B
PCLMULHQHQDQ xmm1, xmm2	0001_0001B

Operation

```
define PCLMUL128(X,Y):                    // helper function
    FOR i ← 0 to 63:
        TMP [ i ] ← X[ 0 ] and Y[ i ]
        FOR j ← 1 to i:
            TMP [ i ] ← TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] ← TMP[ i ]
    FOR i ← 64 to 126:
        TMP [ i ] ← 0
        FOR j ← i - 63 to 63:
            TMP [ i ] ← TMP [ i ] xor (X[ j ] and Y[ i - j ])
        DEST[ i ] ← TMP[ i ]
    DEST[127] ← 0;
    RETURN DEST                           // 128b vector
```

PCLMULQDQ (SSE version)

```
IF Imm8[0] = 0:
    TEMP1 ← SRC1.qword[0]
ELSE:
    TEMP1 ← SRC1.qword[1]
IF Imm8[4] = 0:
    TEMP2 ← SRC2.qword[0]
ELSE:
    TEMP2 ← SRC2.qword[1]
DEST[127:0] ← PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:128] (Unmodified)
```

VPCLMULQDQ (128b and 256b VEX encoded versions)

```
(KL,VL) = (1,128), (2,256)
FOR i= 0 to KL-1:
    IF Imm8[0] = 0:
        TEMP1 ← SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 ← SRC1.xmm[i].qword[1]
    IF Imm8[4] = 0:
        TEMP2 ← SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 ← SRC2.xmm[i].qword[1]
    DEST.xmm[i] ← PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] ← 0
```

VPCLMULQDQ (EVEX encoded version)

```
(KL,VL) = (1,128), (2,256), (4,512)
FOR i = 0 to KL-1:
    IF Imm8[0] = 0:
        TEMP1 ← SRC1.xmm[i].qword[0]
    ELSE:
        TEMP1 ← SRC1.xmm[i].qword[1]
    IF Imm8[4] = 0:
        TEMP2 ← SRC2.xmm[i].qword[0]
    ELSE:
        TEMP2 ← SRC2.xmm[i].qword[1]
    DEST.xmm[i] ← PCLMUL128(TEMP1, TEMP2)
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCLMULQDQ __m256i _mm256_clmulepi64_epi128(__m256i, __m256i, const int);
VPCLMULQDQ __m512i _mm512_clmulepi64_epi128(__m512i, __m512i, const int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VEX-encoded: Exceptions Type 4.

EVEX-encoded: See Exceptions Type E4NF.

VPCOMPRESS — Store Sparse Packed Byte/Word Integer Values into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB m128{k1}, xmm1	A	V/V	AVX512_VBMI2 AVX512VL	Compress up to 128 bits of packed byte values from xmm1 to m128 with writemask k1.
EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB xmm1{k1}{z}, xmm2	B	V/V	AVX512_VBMI2 AVX512VL	Compress up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB m256{k1}, ymm1	A	V/V	AVX512_VBMI2 AVX512VL	Compress up to 256 bits of packed byte values from ymm1 to m256 with writemask k1.
EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB ymm1{k1}{z}, ymm2	B	V/V	AVX512_VBMI2 AVX512VL	Compress up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB m512{k1}, zmm1	A	V/V	AVX512_VBMI2	Compress up to 512 bits of packed byte values from zmm1 to m512 with writemask k1.
EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2	Compress up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.
EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW m128{k1}, xmm1	A	V/V	AVX512_VBMI2 AVX512VL	Compress up to 128 bits of packed word values from xmm1 to m128 with writemask k1.
EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW xmm1{k1}{z}, xmm2	B	V/V	AVX512_VBMI2 AVX512VL	Compress up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW m256{k1}, ymm1	A	V/V	AVX512_VBMI2 AVX512VL	Compress up to 256 bits of packed word values from ymm1 to m256 with writemask k1.
EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW ymm1{k1}{z}, ymm2	B	V/V	AVX512_VBMI2 AVX512VL	Compress up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW m512{k1}, zmm1	A	V/V	AVX512_VBMI2	Compress up to 512 bits of packed word values from zmm1 to m512 with writemask k1.
EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2	Compress up to 512 bits of packed word values from zmm2 to zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compress (stores) up to 64 byte values or 32 word values from the source operand (second operand) to the destination operand (first operand), based on the active elements determined by the writemask operand. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves up to 512 bits of packed byte values from the source operand (second operand) to the destination operand (first operand). This instruction is used to store partial contents of a vector register into a byte vector or single memory location using the active elements in operand writemask.

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.

This instruction supports memory fault suppression.

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

VPCOMPRESSB store form

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] ← SRC.byte[j]
        k ← k +1
```

VPCOMPRESSB reg-reg form

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[k] ← SRC.byte[j]
        k ← k + 1
IF *merging-masking*:
    *DEST[VL-1:k*8] remains unchanged*
    ELSE DEST[VL-1:k*8] ← 0
DEST[MAX_VL-1:VL] ← 0
```

VPCOMPRESSW store form

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] ← SRC.word[j]
        k ← k + 1
```

VPCOMPRESSW reg-reg form

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[k] ← SRC.word[j]
        k ← k + 1
IF *merging-masking*:
    *DEST[VL-1:k*16] remains unchanged*
    ELSE DEST[VL-1:k*16] ← 0
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSB __m128i _mm_mask_compress_epi8(__m128i, __mmask16, __m128i);
VPCOMPRESSB __m128i _mm_maskz_compress_epi8(__mmask16, __m128i);
VPCOMPRESSB __m256i _mm256_mask_compress_epi8(__m256i, __mmask32, __m256i);
VPCOMPRESSB __m256i _mm256_maskz_compress_epi8(__mmask32, __m256i);
VPCOMPRESSB __m512i _mm512_mask_compress_epi8(__m512i, __mmask64, __m512i);
VPCOMPRESSB __m512i _mm512_maskz_compress_epi8(__mmask64, __m512i);
VPCOMPRESSB void _mm_mask_compressstoreu_epi8(void*, __mmask16, __m128i);
VPCOMPRESSB void _mm256_mask_compressstoreu_epi8(void*, __mmask32, __m256i);
VPCOMPRESSB void _mm512_mask_compressstoreu_epi8(void*, __mmask64, __m512i);
VPCOMPRESSW __m128i _mm_mask_compress_epi16(__m128i, __mmask8, __m128i);
VPCOMPRESSW __m128i _mm_maskz_compress_epi16(__mmask8, __m128i);
VPCOMPRESSW __m256i _mm256_mask_compress_epi16(__m256i, __mmask16, __m256i);
VPCOMPRESSW __m256i _mm256_maskz_compress_epi16(__mmask16, __m256i);
VPCOMPRESSW __m512i _mm512_mask_compress_epi16(__m512i, __mmask32, __m512i);
VPCOMPRESSW __m512i _mm512_maskz_compress_epi16(__mmask32, __m512i);
VPCOMPRESSW void _mm_mask_compressstoreu_epi16(void*, __mmask8, __m128i);
VPCOMPRESSW void _mm256_mask_compressstoreu_epi16(void*, __mmask16, __m256i);
VPCOMPRESSW void _mm512_mask_compressstoreu_epi16(void*, __mmask32, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VPDPBUSD — Multiply and Add Unsigned and Signed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W0 50 /r VPDPBUSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 4 pairs of signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1 under writemask k1.
EVEX.DDS.256.66.0F38.W0 50 /r VPDPBUSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 4 pairs of signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1 under writemask k1.
EVEX.DDS.512.66.0F38.W0 50 /r VPDPBUSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 4 pairs of signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result in zmm1 under writemask k1.

Instruction Operand Encoding

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

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand.

This instruction supports memory fault suppression.

Operation

VPDPBUSD dest, src1, src2

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST ← DEST
FOR i ← 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t ← SRC2.dword[0]
        ELSE:
            t ← SRC2.dword[i]
        p1word ← ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word ← ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word ← ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word ← ZERO_EXTEND(SRC1.byte[4*i+3]) * SIGN_EXTEND(t.byte[3])
        DEST.dword[i] ← ORIGDEST.dword[i] + p1word + p2word + p3word + p4word
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE:    // Merge masking, dest element unchanged
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPDPBUSD __m128i _mm_dpbusd_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_mask_dpbusd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSD __m128i _mm_maskz_dpbusd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSD __m256i _mm256_dpbusd_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_mask_dpbusd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSD __m256i _mm256_maskz_dpbusd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSD __m512i _mm512_dpbusd_epi32(__m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_mask_dpbusd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSD __m512i _mm512_maskz_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VPDPBUSDS — Multiply and Add Unsigned and Signed Bytes with Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W0 51 /r VPDPBUSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 4 pairs signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1, under writemask k1.
EVEX.DDS.256.66.0F38.W0 51 /r VPDPBUSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 4 pairs signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1, under writemask k1.
EVEX.DDS.512.66.0F38.W0 51 /r VPDPBUSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 4 pairs signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result, with signed saturation in zmm1, under writemask k1.

Instruction Operand Encoding

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

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand. If the intermediate sum overflows a 32b signed number the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPBUSDS dest, src1, src2

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST ← DEST
FOR i ← 0 TO KL-1:
    IF k1[i] or *no writemask*:
        // Byte elements of SRC1 are zero-extended to 16b and
        // byte elements of SRC2 are sign extended to 16b before multiplication.
        IF SRC2 is memory and EVEX.b == 1:
            t ← SRC2.dword[0]
        ELSE:
            t ← SRC2.dword[i]
        p1word ← ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
        p2word ← ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
        p3word ← ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
        p4word ← ZERO_EXTEND(SRC1.byte[4*i+3]) *SIGN_EXTEND(t.byte[3])
        DEST.dword[i] ← SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
```

```
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE:       // Merge masking, dest element unchanged
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPDPBUSDS __m128i _mm_dpbusds_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_mask_dpbusds_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSDS __m128i _mm_maskz_dpbusds_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSDS __m256i _mm256_dpbusds_epi32(__m256i, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_mask_dpbusds_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_maskz_dpbusds_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSDS __m512i _mm512_dpbusds_epi32(__m512i, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_mask_dpbusds_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_maskz_dpbusds_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VPDPWSSD — Multiply and Add Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W0 52 /r VPDPWSSD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, under writemask k1.
EVEX.DDS.256.66.0F38.W0 52 /r VPDPWSSD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, under writemask k1.
EVEX.DDS.512.66.0F38.W0 52 /r VPDPWSSD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 2 pairs signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, under writemask k1.

Instruction Operand Encoding

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

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand.

This instruction supports memory fault suppression.

Operation

VPDPWSSD dest, src1, src2

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST ← DEST
FOR i ← 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t ← SRC2.dword[0]
        ELSE:
            t ← SRC2.dword[i]
        p1dword ← SRC1.word[2*i] * t.word[0]
        p2dword ← SRC1.word[2*i+1] * t.word[1]
        DEST.dword[i] ← ORIGDEST.dword[i] + p1dword + p2dword
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE:       // Merge masking, dest element unchanged
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPDPWSSD __m128i _mm_dpwssd_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSD __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSD __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSD __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSD __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSD __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSD __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VPDPWSSDS — Multiply and Add Word Integers with Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W0 53 /r VPDPWSSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs of signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation, under writemask k1.
EVEX.DDS.256.66.0F38.W0 53 /r VPDPWSSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512_VNNI AVX512VL	Multiply groups of 2 pairs of signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation, under writemask k1.
EVEX.DDS.512.66.0F38.W0 53 /r VPDPWSSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512_VNNI	Multiply groups of 2 pairs of signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, with signed saturation, under writemask k1.

Instruction Operand Encoding

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

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand. If the intermediate sum overflows a 32b signed number, the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPWSSDS dest, src1, src2

```
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST ← DEST
FOR i ← 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t ← SRC2.dword[0]
        ELSE:
            t ← SRC2.dword[i]
        p1dword ← SRC1.word[2*i] * t.word[0]
        p2dword ← SRC1.word[2*i+1] * t.word[1]
        DEST.dword[i] ← SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
    ELSE IF *zeroing*:
        DEST.dword[i] ← 0
    ELSE:     // Merge masking, dest element unchanged
        DEST.dword[i] ← ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPDPWSSDS __m128i _mm_dpwssds_epi32(__m128i, __m128i, __m128i);
VPDPWSSDS __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSDS __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSDS __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSDS __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSDS __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSDS __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VPEXPAND — Expand Byte/Word Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, m128	A	V/V	AVX512_VBMI2 AVX512VL	Expands up to 128 bits of packed byte values from m128 to xmm1 with writemask k1.
EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, xmm2	B	V/V	AVX512_VBMI2 AVX512VL	Expands up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, m256	A	V/V	AVX512_VBMI2 AVX512VL	Expands up to 256 bits of packed byte values from m256 to ymm1 with writemask k1.
EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, ymm2	B	V/V	AVX512_VBMI2 AVX512VL	Expands up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, m512	A	V/V	AVX512_VBMI2	Expands up to 512 bits of packed byte values from m512 to zmm1 with writemask k1.
EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2	Expands up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.
EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, m128	A	V/V	AVX512_VBMI2 AVX512VL	Expands up to 128 bits of packed word values from m128 to xmm1 with writemask k1.
EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, xmm2	B	V/V	AVX512_VBMI2 AVX512VL	Expands up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.
EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, m256	A	V/V	AVX512_VBMI2 AVX512VL	Expands up to 256 bits of packed word values from m256 to ymm1 with writemask k1.
EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, ymm2	B	V/V	AVX512_VBMI2 AVX512VL	Expands up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.
EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, m512	A	V/V	AVX512_VBMI2	Expands up to 512 bits of packed word values from m512 to zmm1 with writemask k1.
EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, zmm2	B	V/V	AVX512_VBMI2	Expands up to 512 bits of packed byte integer values from zmm2 to zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Expands (loads) up to 64 byte integer values or 32 word integer values from the source operand (memory operand) to the destination operand (register operand), based on the active elements determined by the writemask operand.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves 128, 256 or 512 bits of packed byte integer values from the source operand (memory operand) to the destination operand (register operand). This instruction is used to load from an int8 vector register or memory location while inserting the data into sparse elements of destination vector register using the active elements pointed out by the operand writemask.

This instruction supports memory fault suppression.

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

VPEXPANDB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.byte[j] ← SRC.byte[k];
        k ← k + 1
        ELSE:
            IF *merging-masking*:
                *DEST.byte[j] remains unchanged*
                ELSE:                       ; zeroing-masking
                    DEST.byte[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

VPEXPANDW

```
(KL, VL) = (8,128), (16,256), (32, 512)
k ← 0
FOR j ← 0 TO KL-1:
    IF k1[j] OR *no writemask*:
        DEST.word[j] ← SRC.word[k];
        k ← k + 1
        ELSE:
            IF *merging-masking*:
                *DEST.word[j] remains unchanged*
                ELSE:                       ; zeroing-masking
                    DEST.word[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPAND __m128i _mm_mask_expand_epi8(__m128i, __mmask16, __m128i);
VPEXPAND __m128i _mm_maskz_expand_epi8(__mmask16, __m128i);
VPEXPAND __m128i _mm_mask_expandloadu_epi8(__m128i, __mmask16, const void*);
VPEXPAND __m128i _mm_maskz_expandloadu_epi8(__mmask16, const void*);
VPEXPAND __m256i _mm256_mask_expand_epi8(__m256i, __mmask32, __m256i);
VPEXPAND __m256i _mm256_maskz_expand_epi8(__mmask32, __m256i);
VPEXPAND __m256i _mm256_mask_expandloadu_epi8(__m256i, __mmask32, const void*);
VPEXPAND __m256i _mm256_maskz_expandloadu_epi8(__mmask32, const void*);
VPEXPAND __m512i _mm512_mask_expand_epi8(__m512i, __mmask64, __m512i);
VPEXPAND __m512i _mm512_maskz_expand_epi8(__mmask64, __m512i);
VPEXPAND __m512i _mm512_mask_expandloadu_epi8(__m512i, __mmask64, const void*);
VPEXPAND __m512i _mm512_maskz_expandloadu_epi8(__mmask64, const void*);
VPEXPANDW __m128i _mm_mask_expand_epi16(__m128i, __mmask8, __m128i);
VPEXPANDW __m128i _mm_maskz_expand_epi16(__mmask8, __m128i);
VPEXPANDW __m128i _mm_mask_expandloadu_epi16(__m128i, __mmask8, const void*);
VPEXPANDW __m128i _mm_maskz_expandloadu_epi16(__mmask8, const void *);
VPEXPANDW __m256i _mm256_mask_expand_epi16(__m256i, __mmask16, __m256i);
VPEXPANDW __m256i _mm256_maskz_expand_epi16(__mmask16, __m256i);
VPEXPANDW __m256i _mm256_mask_expandloadu_epi16(__m256i, __mmask16, const void*);
VPEXPANDW __m256i _mm256_maskz_expandloadu_epi16(__mmask16, const void*);
VPEXPANDW __m512i _mm512_mask_expand_epi16(__m512i, __mmask32, __m512i);
VPEXPANDW __m512i _mm512_maskz_expand_epi16(__mmask32, __m512i);
VPEXPANDW __m512i _mm512_mask_expandloadu_epi16(__m512i, __mmask32, const void*);
VPEXPANDW __m512i _mm512_maskz_expandloadu_epi16(__mmask32, const void*);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4.

VPOPCNT — Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 54 /r VPOPCNTB xmm1{k1}{z}, xmm2/m128	A	V/V	AVX512_BITALG AVX512VL	Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W0 54 /r VPOPCNTB ymm1{k1}{z}, ymm2/m256	A	V/V	AVX512_BITALG AVX512VL	Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W0 54 /r VPOPCNTB zmm1{k1}{z}, zmm2/m512	A	V/V	AVX512_BITALG	Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W1 54 /r VPOPCNTW xmm1{k1}{z}, xmm2/m128	A	V/V	AVX512_BITALG AVX512VL	Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W1 54 /r VPOPCNTW ymm1{k1}{z}, ymm2/m256	A	V/V	AVX512_BITALG AVX512VL	Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W1 54 /r VPOPCNTW zmm1{k1}{z}, zmm2/m512	A	V/V	AVX512_BITALG	Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W0 55 /r VPOPCNTD xmm1{k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512_VPOPCNTDQ AVX512VL	Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W0 55 /r VPOPCNTD ymm1{k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512_VPOPCNTDQ AVX512VL	Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W0 55 /r VPOPCNTD zmm1{k1}{z}, zmm2/m512/m32bcst	B	V/V	AVX512_VPOPCNTDQ	Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1.
EVEX.128.66.0F38.W1 55 /r VPOPCNTQ xmm1{k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512_VPOPCNTDQ AVX512VL	Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.
EVEX.256.66.0F38.W1 55 /r VPOPCNTQ ymm1{k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512_VPOPCNTDQ AVX512VL	Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.
EVEX.512.66.0F38.W1 55 /r VPOPCNTQ zmm1{k1}{z}, zmm2/m512/m64bcst	B	V/V	AVX512_VPOPCNTDQ	Counts the number of bits set to one in zmm2/m512/m64bcst and puts the result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction counts the number of bits set to one in each byte, word, dword or qword element of its source (e.g., zmm2 or memory) and places the results in the destination register (zmm1). This instruction supports memory fault suppression.

Operation

VPOPCNTB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.byte[j] ← POPCNT(SRC.byte[j])
    ELSE IF *merging-masking*:
        *DEST.byte[j] remains unchanged*
    ELSE:
        DEST.byte[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

VPOPCNTW

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] ← POPCNT(SRC.word[j])
    ELSE IF *merging-masking*:
        *DEST.word[j] remains unchanged*
    ELSE:
        DEST.word[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

VPOPCNTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t ← SRC.dword[0]
        ELSE:
            t ← SRC.dword[j]
        DEST.dword[j] ← POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..dword[j] remains unchanged*
    ELSE:
        DEST..dword[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

VPOPCNTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        IF SRC is broadcast memop:
            t ← SRC.qword[0]
        ELSE:
            t ← SRC.qword[j]
        DEST.qword[j] ← POPCNT(t)
    ELSE IF *merging-masking*:
        *DEST..qword[j] remains unchanged*
    ELSE:
        DEST..qword[j] ← 0
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPOPCNTW __m128i _mm_popcnt_epi16(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi16(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi16(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi16(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi16(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi16(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi16(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi16(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i _mm512_maskz_popcnt_epi16(__mmask32, __m512i);
VPOPCNTQ __m128i _mm_popcnt_epi64(__m128i);
VPOPCNTQ __m128i _mm_mask_popcnt_epi64(__m128i, __mmask8, __m128i);
VPOPCNTQ __m128i _mm_maskz_popcnt_epi64(__mmask8, __m128i);
VPOPCNTQ __m256i _mm256_popcnt_epi64(__m256i);
VPOPCNTQ __m256i _mm256_mask_popcnt_epi64(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i _mm256_maskz_popcnt_epi64(__mmask8, __m256i);
VPOPCNTQ __m512i _mm512_popcnt_epi64(__m512i);
VPOPCNTQ __m512i _mm512_mask_popcnt_epi64(__m512i, __mmask8, __m512i);
VPOPCNTQ __m512i _mm512_maskz_popcnt_epi64(__mmask8, __m512i);
VPOPCNTD __m128i _mm_popcnt_epi32(__m128i);
VPOPCNTD __m128i _mm_mask_popcnt_epi32(__m128i, __mmask8, __m128i);
VPOPCNTD __m128i _mm_maskz_popcnt_epi32(__mmask8, __m128i);
VPOPCNTD __m256i _mm256_popcnt_epi32(__m256i);
VPOPCNTD __m256i _mm256_mask_popcnt_epi32(__m256i, __mmask8, __m256i);
VPOPCNTD __m256i _mm256_maskz_popcnt_epi32(__mmask8, __m256i);
VPOPCNTD __m512i _mm512_popcnt_epi32(__m512i);
VPOPCNTD __m512i _mm512_mask_popcnt_epi32(__m512i, __mmask16, __m512i);
VPOPCNTD __m512i _mm512_maskz_popcnt_epi32(__mmask16, __m512i);
VPOPCNTB __m128i _mm_popcnt_epi8(__m128i);
VPOPCNTB __m128i _mm_mask_popcnt_epi8(__m128i, __mmask16, __m128i);
VPOPCNTB __m128i _mm_maskz_popcnt_epi8(__mmask16, __m128i);
VPOPCNTB __m256i _mm256_popcnt_epi8(__m256i);
VPOPCNTB __m256i _mm256_mask_popcnt_epi8(__m256i, __mmask32, __m256i);
VPOPCNTB __m256i _mm256_maskz_popcnt_epi8(__mmask32, __m256i);
VPOPCNTB __m512i _mm512_popcnt_epi8(__m512i);
VPOPCNTB __m512i _mm512_mask_popcnt_epi8(__m512i, __mmask64, __m512i);
VPOPCNTB __m512i _mm512_maskz_popcnt_epi8(__mmask64, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4.

VPSHLD — Concatenate and Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 70 /r /ib VPSHLDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.NDS.256.66.0F3A.W1 70 /r /ib VPSHLDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.NDS.512.66.0F3A.W1 70 /r /ib VPSHLDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512_VBMI2	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.
EVEX.NDS.128.66.0F3A.W0 71 /r /ib VPSHLDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.NDS.256.66.0F3A.W0 71 /r /ib VPSHLDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.NDS.512.66.0F3A.W0 71 /r /ib VPSHLDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	B	V/V	AVX512_VBMI2	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.
EVEX.NDS.128.66.0F3A.W1 71 /r /ib VPSHLDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.
EVEX.NDS.256.66.0F3A.W1 71 /r /ib VPSHLDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.
EVEX.NDS.512.66.0F3A.W1 71 /r /ib VPSHLDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	B	V/V	AVX512_VBMI2	Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	imm8 (r)

Description

Concatenate packed data, extract result shifted to the left by constant value.

This instruction supports memory fault suppression.

Operation

VPSHLDW DEST, SRC2, SRC3, imm8

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(SRC2.word[j], SRC3.word[j]) << (imm8 & 15)
        DEST.word[j] ← tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] ← 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VPSHLDD DEST, SRC2, SRC3, imm8

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.dword[0]
    ELSE:
        tsrc3 ← SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(SRC2.dword[j], tsrc3) << (imm8 & 31)
        DEST.dword[j] ← tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] ← 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VPSHLDQ DEST, SRC2, SRC3, imm8

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.qword[0]
    ELSE:
        tsrc3 ← SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(SRC2.qword[j], tsrc3) << (imm8 & 63)
        DEST.qword[j] ← tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] ← 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHLDD __m128i _mm_shldi_epi32(__m128i, __m128i, int);
VPSHLDD __m128i _mm_mask_shldi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDD __m128i _mm_maskz_shldi_epi32(__mmask8, __m128i, __m128i, int);
VPSHLDD __m256i _mm256_shldi_epi32(__m256i, __m256i, int);
VPSHLDD __m256i _mm256_mask_shldi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDD __m256i _mm256_maskz_shldi_epi32(__mmask8, __m256i, __m256i, int);
VPSHLDD __m512i _mm512_shldi_epi32(__m512i, __m512i, int);
VPSHLDD __m512i _mm512_mask_shldi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHLDD __m512i _mm512_maskz_shldi_epi32(__mmask16, __m512i, __m512i, int);
VPSHLDQ __m128i _mm_shldi_epi64(__m128i, __m128i, int);
VPSHLDQ __m128i _mm_mask_shldi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDQ __m128i _mm_maskz_shldi_epi64(__mmask8, __m128i, __m128i, int);
VPSHLDQ __m256i _mm256_shldi_epi64(__m256i, __m256i, int);
VPSHLDQ __m256i _mm256_mask_shldi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDQ __m256i _mm256_maskz_shldi_epi64(__mmask8, __m256i, __m256i, int);
VPSHLDQ __m512i _mm512_shldi_epi64(__m512i, __m512i, int);
VPSHLDQ __m512i _mm512_mask_shldi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHLDQ __m512i _mm512_maskz_shldi_epi64(__mmask8, __m512i, __m512i, int);
VPSHLDW __m128i _mm_shldi_epi16(__m128i, __m128i, int);
VPSHLDW __m128i _mm_mask_shldi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDW __m128i _mm_maskz_shldi_epi16(__mmask8, __m128i, __m128i, int);
VPSHLDW __m256i _mm256_shldi_epi16(__m256i, __m256i, int);
VPSHLDW __m256i _mm256_mask_shldi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHLDW __m256i _mm256_maskz_shldi_epi16(__mmask16, __m256i, __m256i, int);
VPSHLDW __m512i _mm512_shldi_epi16(__m512i, __m512i, int);
VPSHLDW __m512i _mm512_mask_shldi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHLDW __m512i _mm512_maskz_shldi_epi16(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4.

VPSHLDV — Concatenate and Variable Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 70 /r VPSHLDVW xmm1{k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512_VBMI2 AVX512VL	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.DDS.256.66.0F38.W1 70 /r VPSHLDVW ymm1{k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512_VBMI2 AVX512VL	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.DDS.512.66.0F38.W1 70 /r VPSHLDVW zmm1{k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI2	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.
EVEX.DDS.128.66.0F38.W0 71 /r VPSHLDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.DDS.256.66.0F38.W0 71 /r VPSHLDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.DDS.512.66.0F38.W0 71 /r VPSHLDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VBMI2	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.
EVEX.DDS.128.66.0F38.W1 71 /r VPSHLDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.
EVEX.DDS.256.66.0F38.W1 71 /r VPSHLDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.
EVEX.DDS.512.66.0F38.W1 71 /r VPSHLDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_VBMI2	Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1.

Instruction Operand Encoding

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

Description

Concatenate packed data, extract result shifted to the left by variable value.

This instruction supports memory fault suppression.

Operation

```
FUNCTION concat(a,b):
    IF words:
        d.word[1] ← a
        d.word[0] ← b
        return d
    ELSE IF dwords:
        q.dword[1] ← a
        q.dword[0] ← b
        return q
    ELSE IF qwords:
        o.qword[1] ← a
        o.qword[0] ← b
        return o
```

VPSHLDVW DEST, SRC2, SRC3

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(DEST.word[j], SRC2.word[j]) << (SRC3.word[j] & 15)
        DEST.word[j] ← tmp.word[1]
    ELSE IF *zeroing*:
        DEST.word[j] ← 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VPSHLDVD DEST, SRC2, SRC3

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.dword[0]
    ELSE:
        tsrc3 ← SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(DEST.dword[j], SRC2.dword[j]) << (tsrc3 & 31)
        DEST.dword[j] ← tmp.dword[1]
    ELSE IF *zeroing*:
        DEST.dword[j] ← 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VPSHLDVQ DEST, SRC2, SRC3

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.qword[0]
    ELSE:
        tsrc3 ← SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp ← concat(DEST.qword[j], SRC2.qword[j]) << (tsrc3 & 63)
        DEST.qword[j] ← tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] ← 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVQ __m512i _mm512_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_mask_shldv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_maskz_shldv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVW __m512i _mm512_shldv_epi16(__m512i, __m512i, __m512i);
VPSHLDVW __m512i _mm512_mask_shldv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHLDVW __m512i _mm512_maskz_shldv_epi16(__mmask32, __m512i, __m512i, __m512i);
VPSHLDVD __m128i _mm_shldv_epi32(__m128i, __m128i, __m128i);
VPSHLDVD __m128i _mm_mask_shldv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVD __m128i _mm_maskz_shldv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVD __m256i _mm256_shldv_epi32(__m256i, __m256i, __m256i);
VPSHLDVD __m256i _mm256_mask_shldv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHLDVD __m256i _mm256_maskz_shldv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHLDVD __m512i _mm512_shldv_epi32(__m512i, __m512i, __m512i);
VPSHLDVD __m512i _mm512_mask_shldv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHLDVD __m512i _mm512_maskz_shldv_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4.

VPSHRD — Concatenate and Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 72 /r /ib VPSHRDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.NDS.256.66.0F3A.W1 72 /r /ib VPSHRDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.NDS.512.66.0F3A.W1 72 /r /ib VPSHRDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8	A	V/V	AVX512_VBMI2	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.
EVEX.NDS.128.66.0F3A.W0 73 /r /ib VPSHRDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.NDS.256.66.0F3A.W0 73 /r /ib VPSHRDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.NDS.512.66.0F3A.W0 73 /r /ib VPSHRDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	B	V/V	AVX512_VBMI2	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.
EVEX.NDS.128.66.0F3A.W1 73 /r /ib VPSHRDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.
EVEX.NDS.256.66.0F3A.W1 73 /r /ib VPSHRDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.
EVEX.NDS.512.66.0F3A.W1 73 /r /ib VPSHRDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	B	V/V	AVX512_VBMI2	Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	imm8 (r)
B	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	imm8 (r)

Description

Concatenate packed data, extract result shifted to the right by constant value.

This instruction supports memory fault suppression.

Operation

VPSHRDW DEST, SRC2, SRC3, imm8

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] ← concat(SRC3.word[j], SRC2.word[j]) >> (imm8 & 15)
    ELSE IF *zeroing*:
        DEST.word[j] ← 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VPSHRDD DEST, SRC2, SRC3, imm8

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.dword[0]
    ELSE:
        tsrc3 ← SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] ← concat(tsrc3, SRC2.dword[j]) >> (imm8 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] ← 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VPSHRDQ DEST, SRC2, SRC3, imm8

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.qword[0]
    ELSE:
        tsrc3 ← SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] ← concat(tsrc3, SRC2.qword[j]) >> (imm8 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] ← 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHRDQ __m128i _mm_shrdi_epi64(__m128i, __m128i, int);
VPSHRDQ __m128i _mm_mask_shrdi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDQ __m128i _mm_maskz_shrdi_epi64(__mmask8, __m128i, __m128i, int);
VPSHRDQ __m256i _mm256_shrdi_epi64(__m256i, __m256i, int);
VPSHRDQ __m256i _mm256_mask_shrdi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDQ __m256i _mm256_maskz_shrdi_epi64(__mmask8, __m256i, __m256i, int);
VPSHRDQ __m512i _mm512_shrdi_epi64(__m512i, __m512i, int);
VPSHRDQ __m512i _mm512_mask_shrdi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHRDQ __m512i _mm512_maskz_shrdi_epi64(__mmask8, __m512i, __m512i, int);
VPSHRDD __m128i _mm_shrdi_epi32(__m128i, __m128i, int);
VPSHRDD __m128i _mm_mask_shrdi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDD __m128i _mm_maskz_shrdi_epi32(__mmask8, __m128i, __m128i, int);
VPSHRDD __m256i _mm256_shrdi_epi32(__m256i, __m256i, int);
VPSHRDD __m256i _mm256_mask_shrdi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDD __m256i _mm256_maskz_shrdi_epi32(__mmask8, __m256i, __m256i, int);
VPSHRDD __m512i _mm512_shrdi_epi32(__m512i, __m512i, int);
VPSHRDD __m512i _mm512_mask_shrdi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHRDD __m512i _mm512_maskz_shrdi_epi32(__mmask16, __m512i, __m512i, int);
VPSHRDW __m128i _mm_shrdi_epi16(__m128i, __m128i, int);
VPSHRDW __m128i _mm_mask_shrdi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDW __m128i _mm_maskz_shrdi_epi16(__mmask8, __m128i, __m128i, int);
VPSHRDW __m256i _mm256_shrdi_epi16(__m256i, __m256i, int);
VPSHRDW __m256i _mm256_mask_shrdi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHRDW __m256i _mm256_maskz_shrdi_epi16(__mmask16, __m256i, __m256i, int);
VPSHRDW __m512i _mm512_shrdi_epi16(__m512i, __m512i, int);
VPSHRDW __m512i _mm512_mask_shrdi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHRDW __m512i _mm512_maskz_shrdi_epi16(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4.

VPSHRDV — Concatenate and Variable Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 72 /r VPSHRDVW xmm1{k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512_VBMI2 AVX512VL	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.DDS.256.66.0F38.W1 72 /r VPSHRDVW ymm1{k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512_VBMI2 AVX512VL	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.DDS.512.66.0F38.W1 72 /r VPSHRDVW zmm1{k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI2	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.
EVEX.DDS.128.66.0F38.W0 73 /r VPSHRDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.DDS.256.66.0F38.W0 73 /r VPSHRDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.DDS.512.66.0F38.W0 73 /r VPSHRDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512_VBMI2	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.
EVEX.DDS.128.66.0F38.W1 73 /r VPSHRDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.
EVEX.DDS.256.66.0F38.W1 73 /r VPSHRDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512_VBMI2 AVX512VL	Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.
EVEX.DDS.512.66.0F38.W1 73 /r VPSHRDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512_VBMI2	Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1.

Instruction Operand Encoding

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

Description

Concatenate packed data, extract result shifted to the right by variable value.

This instruction supports memory fault suppression.

Operation

VPSHRDVW DEST, SRC2, SRC3

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1:
    IF MaskBit(j) OR *no writemask*:
        DEST.word[j] ← concat(SRC2.word[j], DEST.word[j]) >> (SRC3.word[j] & 15)
    ELSE IF *zeroing*:
        DEST.word[j] ← 0
    *ELSE DEST.word[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VPSHRDVD DEST, SRC2, SRC3

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.dword[0]
    ELSE:
        tsrc3 ← SRC3.dword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.dword[j] ← concat(SRC2.dword[j], DEST.dword[j]) >> (tsrc3 & 31)
    ELSE IF *zeroing*:
        DEST.dword[j] ← 0
    *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

VPSHRDVQ DEST, SRC2, SRC3

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 ← SRC3.qword[0]
    ELSE:
        tsrc3 ← SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        DEST.qword[j] ← concat(SRC2.qword[j], DEST.qword[j]) >> (tsrc3 & 63)
    ELSE IF *zeroing*:
        DEST.qword[j] ← 0
    *ELSE DEST.qword[j] remains unchanged*
DEST[MAX_VL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHRDVQ __m128i _mm_shrdv_epi64(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i _mm_mask_shrdv_epi64(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i _mm_maskz_shrdv_epi64(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i _mm256_shrdv_epi64(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_mask_shrdv_epi64(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_maskz_shrdv_epi64(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i _mm512_shrdv_epi64(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_mask_shrdv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_maskz_shrdv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHRDVD __m128i _mm_shrdv_epi32(__m128i, __m128i, __m128i);
VPSHRDVD __m128i _mm_mask_shrdv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVD __m128i _mm_maskz_shrdv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVD __m256i _mm256_shrdv_epi32(__m256i, __m256i, __m256i);
VPSHRDVD __m256i _mm256_mask_shrdv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVD __m256i _mm256_maskz_shrdv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVD __m512i _mm512_shrdv_epi32(__m512i, __m512i, __m512i);
VPSHRDVD __m512i _mm512_mask_shrdv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHRDVD __m512i _mm512_maskz_shrdv_epi32(__mmask16, __m512i, __m512i, __m512i);
VPSHRDVW __m128i _mm_shrdv_epi16(__m128i, __m128i, __m128i);
VPSHRDVW __m128i _mm_mask_shrdv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVW __m128i _mm_maskz_shrdv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVW __m256i _mm256_shrdv_epi16(__m256i, __m256i, __m256i);
VPSHRDVW __m256i _mm256_mask_shrdv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHRDVW __m256i _mm256_maskz_shrdv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHRDVW __m512i _mm512_shrdv_epi16(__m512i, __m512i, __m512i);
VPSHRDVW __m512i _mm512_mask_shrdv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHRDVW __m512i _mm512_maskz_shrdv_epi16(__mmask32, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4.

VPSHUFBITQMB — Shuffle Bits from Quadword Elements Using Byte Indexes into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, xmm2, xmm3/m128	A	V/V	AVX512_BITALG AVX512VL	Extract values in xmm2 using control bits of xmm3/m128 with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, ymm2, ymm3/m256	A	V/V	AVX512_BITALG AVX512VL	Extract values in ymm2 using control bits of ymm3/m256 with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, zmm2, zmm3/m512	A	V/V	AVX512_BITALG	Extract values in zmm2 using control bits of zmm3/m512 with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

The VPSHUFBITQMB instruction performs a bit gather select using second source as control and first source as data. Each bit uses 6 control bits (2nd source operand) to select which data bit is going to be gathered (first source operand). A given bit can only access 64 different bits of data (first 64 destination bits can access first 64 data bits, second 64 destination bits can access second 64 data bits, etc.).

Control data for each output bit is stored in 8 bit elements of SRC2, but only the 6 least significant bits of each element are used.

This instruction uses write masking (zeroing only). This instruction supports memory fault suppression.

The first source operand is a ZMM register. The second source operand is a ZMM register or a memory location. The destination operand is a mask register.

Operation

VPSHUFBITQMB DEST, SRC1, SRC2

```
(KL, VL) = (16,128), (32,256), (64, 512)
FOR i ← 0 TO KL/8-1:                //Qword
    FOR j ← 0 to 7:                 // Byte
        IF k2[i*8+j] or *no writemask*:
            m ← SRC2.qword[i].byte[j] & 0x3F
            k1[i*8+j] ← SRC1.qword[i].bit[m]
        ELSE:
            k1[i*8+j] ← 0
k1[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFBITQMB __mmask16 _mm_bitshuffle_epi64_mask(__m128i, __m128i);
VPSHUFBITQMB __mmask16 _mm_mask_bitshuffle_epi64_mask(__mmask16, __m128i, __m128i);
VPSHUFBITQMB __mmask32 _mm256_bitshuffle_epi64_mask(__m256i, __m256i);
VPSHUFBITQMB __mmask32 _mm256_mask_bitshuffle_epi64_mask(__mmask32, __m256i, __m256i);
VPSHUFBITQMB __mmask64 _mm512_bitshuffle_epi64_mask(__m512i, __m512i);
VPSHUFBITQMB __mmask64 _mm512_mask_bitshuffle_epi64_mask(__mmask64, __m512i, __m512i);

WBNOINVD—Write Back and Do Not Invalidate Cache

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 09 WBNOINVD	A	V/V	WBNOINVD	Write back and do not flush internal caches; initiate writing-back without flushing of external caches.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA	NA

Description

The WBNOINVD instruction writes back all modified cache lines in the processor's internal cache to main memory but does not invalidate (flush) the internal caches.

After executing this instruction, the processor does not wait for the external caches to complete their write-back operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back signal. The amount of time or cycles for WBNOINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBNOINVD instruction can have an impact on logical processor interrupt/event response time.

The WBNOINVD 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 WBNOINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors.

Operation

```
WriteBack(InternalCaches);
Continue; (* Continue execution *)
```

Intel C/C++ Compiler Intrinsic Equivalent

WBNOINVD void _wbnoinvd(void);

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)	WBNOINVD 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.