

## V4FMADDPS/V4FNMADDPS – Packed Single-Precision Floating-Point Fused Multiply-Add (4-iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.512.F2.0F38.W0 9A /r V4FMADDPS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.
EVEX.DDS.512.F2.0F38.W0 AA /r V4FNMADDPS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply and negate packed single-precision floating-point values from source register block indicated by zmm2 by values from m128 and accumulate the result in zmm1.

### Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	T1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction computes 4 sequential packed fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any of the 16 lowest significant mask bits is set to 1 or if a “no masking” encoding is used.

The tuple type T1\_4X implies that 4 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA (fused multiply and add) boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

**Operation**

src\_reg\_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

define NFMA\_PS(kl, vl, dest, k1, msrc, regs\_loaded, src\_base, posneg):

tmpdest ← dest

// reg[] is an array representing the SIMD register file.

for j ← 0 to regs\_loaded-1:

for i ← 0 to kl-1:

if k1[i] or \*no writemask\*:

if posneg = 0:

tmpdest.single[i] ← RoundFPControl\_MXCSR(tmpdest.single[i] - reg[src\_base + j].single[i] \* msrc.single[j])

else:

tmpdest.single[i] ← RoundFPControl\_MXCSR(tmpdest.single[i] + reg[src\_base + j].single[i] \* msrc.single[j])

else if \*zeroing\*:

tmpdest.single[i] ← 0

dest ← tmpdst

dest[MAX\_VL-1:VL] ← 0

V4FMADDPS and V4FNMADDPS dest{k1}, src1, msrc (AVX512)

kl, vl = (16, 512)

regs\_loaded ← 4

src\_base ← src\_reg\_id & ~3 // for src1 operand

posneg ← 0 if negative form, 1 otherwise

NFMA\_PS(kl, vl, dest, k1, msrc, regs\_loaded, src\_base, posneg)

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

V4FMADDPS \_\_m512 \_\_mm512\_4fmadd\_ps(\_\_m512, \_\_m512x4, \_\_m128 \*);

V4FMADDPS \_\_m512 \_\_mm512\_mask\_4fmadd\_ps(\_\_m512, \_\_mmask16, \_\_m512x4, \_\_m128 \*);

V4FMADDPS \_\_m512 \_\_mm512\_maskz\_4fmadd\_ps(\_\_mmask16, \_\_m512, \_\_m512x4, \_\_m128 \*);

V4FNMADDPS \_\_m512 \_\_mm512\_4fnmadd\_ps(\_\_m512, \_\_m512x4, \_\_m128 \*);

V4FNMADDPS \_\_m512 \_\_mm512\_mask\_4fnmadd\_ps(\_\_m512, \_\_mmask16, \_\_m512x4, \_\_m128 \*);

V4FNMADDPS \_\_m512 \_\_mm512\_maskz\_4fnmadd\_ps(\_\_mmask16, \_\_m512, \_\_m512x4, \_\_m128 \*);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

See Type E2; additionally

#UD If the EVEX broadcast bit is set to 1.

#UD If the MODRM.mod = 0b11.

## V4FMADDSS/V4FNMADDSS —Scalar Single-Precision Floating-Point Fused Multiply-Add (4-iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.LLIG.F2.0F38.W0 9B /r V4FMADDSS xmm1{k1}{z}, xmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.
EVEX.DDS.LLIG.F2.0F38.W0 AB /r V4FNMADDSS xmm1{k1}{z}, xmm2+3, m128	A	V/V	AVX512_4FMAPS	Multiply and negate scalar single-precision floating-point values from source register block indicated by xmm2 by values from m128 and accumulate the result in xmm1.

### Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	T1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction computes 4 sequential scalar fused single-precision floating-point multiply-add instructions with a sequentially selected memory operand in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if the least significant mask bit is set to 1 or if a “no masking” encoding is used.

The tuple type T1\_4X implies that 4 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

Rounding is performed at every FMA boundary. Exceptions are also taken sequentially. Pre- and post-computational exceptions of the first FMA take priority over the pre- and post-computational exceptions of the second FMA, etc.

### Operation

src\_reg\_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

```

define NFMA_SS(vl, dest, k1, msrc, regs_loaded, src_base, posneg):
    tmpdest ← dest
    // reg[] is an array representing the SIMD register file.
    if k1[0] or *no writemask*:
        for j ← 0 to regs_loaded - 1:
            if posneg = 0:
                tmpdest.single[0] ← RoundFPControl_MXCSR(tmpdest.single[0] - reg[src_base + j].single[0] * msrc.single[j])
            else:
                tmpdest.single[0] ← RoundFPControl_MXCSR(tmpdest.single[0] + reg[src_base + j].single[0] * msrc.single[j])
    else if *zeroing*:
        tmpdest.single[0] ← 0
    dest ← tmpdest
    dest[MAX_VL-1:VL] ← 0

```

V4FMADDSS and V4FNMADDSS dest{k1}, src1, msrc (AVX512)  
 vl = 128

regs\_loaded  $\leftarrow$  4  
 src\_base  $\leftarrow$  src\_reg\_id & ~3 // for src1 operand  
 posneg  $\leftarrow$  0 if negative form, 1 otherwise  
 NFMA\_SS(vl, dest, k1, msrc, regs\_loaded, src\_base, posneg)

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

V4FMADDSS \_\_m128 \_mm\_4fmadd\_ss(\_\_m128, \_\_m128x4, \_\_m128 \*);  
 V4FMADDSS \_\_m128 \_mm\_mask\_4fmadd\_ss(\_\_m128, \_\_mmask8, \_\_m128x4, \_\_m128 \*);  
 V4FMADDSS \_\_m128 \_mm\_maskz\_4fmadd\_ss(\_\_mmask8, \_\_m128, \_\_m128x4, \_\_m128 \*);  
 V4FNMADDSS \_\_m128 \_mm\_4fnmadd\_ss(\_\_m128, \_\_m128x4, \_\_m128 \*);  
 V4FNMADDSS \_\_m128 \_mm\_mask\_4fnmadd\_ss(\_\_m128, \_\_mmask8, \_\_m128x4, \_\_m128 \*);  
 V4FNMADDSS \_\_m128 \_mm\_maskz\_4fnmadd\_ss(\_\_mmask8, \_\_m128, \_\_m128x4, \_\_m128 \*);

#### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

#### Other Exceptions

See Type E2; additionally

#UD                      If the EVEX broadcast bit is set to 1.  
 #UD                      If the MODRM.mod = 0b11.

## VP4DPWSSD — Dot Product of Signed Words with Dword Accumulation (4-iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEEX.DDS.512.F2.0F38.W0 52 /r VP4DPWSSD zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4VNNIW	Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate resulting signed dwords in zmm1.

### Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	T1_4X	ModRM:reg (r, w)	EVEEX.vvvv (r)	ModRM:r/m (r)	NA

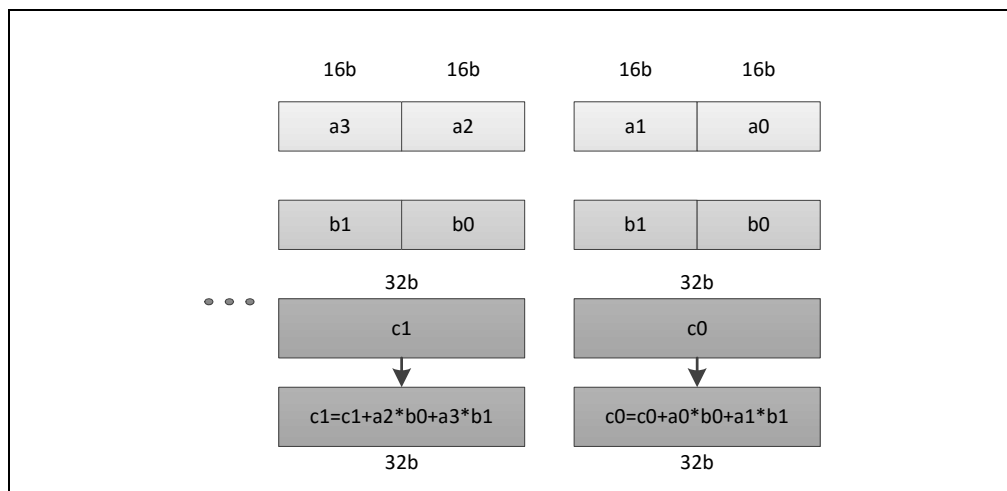
### Description

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation; see Figure 5-1 below. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a “no masking” encoding is used.

The tuple type T1\_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.



**Figure 5-1. Register Source-Block Dot Product of Two Signed Word Operands with Doubleword Accumulation<sup>1</sup>**

#### NOTES:

1. For illustration purposes, one source-block dot product instance is shown out of the four.

**Operation**

src\_reg\_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

VP4DPWSSD dest, src1, src2

(KL,VL) = (16,512)

$N \leftarrow 4$

ORIGDEST  $\leftarrow$  DEST

src\_base  $\leftarrow$  src\_reg\_id &  $\sim(N-1)$  // for src1 operand

FOR i  $\leftarrow$  0 to KL-1:

IF k1[i] or \*no writemask\*:

FOR m  $\leftarrow$  0 to N-1:

t  $\leftarrow$  SRC2.dword[m]

p1dword  $\leftarrow$  reg[src\_base+m].word[2\*i] \* t.word[0]

p2dword  $\leftarrow$  reg[src\_base+m].word[2\*i+1] \* t.word[1]

DEST.dword[i]  $\leftarrow$  DEST.dword[i] + p1dword + p2dword

ELSE IF \*zeroing\*:

DEST.dword[i]  $\leftarrow$  0

ELSE

DEST.dword[i]  $\leftarrow$  ORIGDEST.dword[i]

DEST[MAX\_VL-1:VL]  $\leftarrow$  0

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

VP4DPWSSD \_\_m512i \_mm512\_4dpwssd\_epi32(\_\_m512i, \_\_m512ix4, \_\_m128i \*);

VP4DPWSSD \_\_m512i \_mm512\_mask\_4dpwssd\_epi32(\_\_m512i, \_\_mmask16, \_\_m512ix4, \_\_m128i \*);

VP4DPWSSD \_\_m512i \_mm512\_maskz\_4dpwssd\_epi32(\_\_mmask16, \_\_m512i, \_\_m512ix4, \_\_m128i \*);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Type E4; additionally

#UD If the EVEX broadcast bit is set to 1.

#UD If the MODRM.mod = 0b11.

## VP4DPWSSDS — Dot Product of Signed Words with Dword Accumulation and Saturation (4-iterations)

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.512.F2.0F38.W0 53 /r VP4DPWSSDS zmm1{k1}{z}, zmm2+3, m128	A	V/V	AVX512_4VNNIW	Multiply signed words from source register block indicated by zmm2 by signed words from m128 and accumulate the resulting dword results with signed saturation in zmm1.

### Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	T1_4X	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

This instruction computes 4 sequential register source-block dot-products of two signed word operands with doubleword accumulation and signed saturation. The memory operand is sequentially selected in each of the four steps.

In the above box, the notation of “+3” is used to denote that the instruction accesses 4 source registers based on that operand; sources are consecutive, start in a multiple of 4 boundary, and contain the encoded register operand.

This instruction supports memory fault suppression. The entire memory operand is loaded if any bit of the lowest 16-bits of the mask is set to 1 or if a “no masking” encoding is used.

The tuple type T1\_4X implies that four 32-bit elements (16 bytes) are referenced by the memory operation portion of this instruction.

### Operation

src\_reg\_id is the 5 bit index of the vector register specified in the instruction as the src1 register.

VP4DPWSSDS dest, src1, src2

(KL,VL) = (16,512)

N ← 4

ORIGDEST ← DEST

src\_base ← src\_reg\_id & ~ (N-1) // for src1 operand

FOR i ← 0 to KL-1:

IF k1[i] or \*no writemask\*:

FOR m ← 0 to N-1:

t ← SRC2.dword[m]

p1dword ← reg[src\_base+m].word[2\*i] \* t.word[0]

p2dword ← reg[src\_base+m].word[2\*i+1] \* t.word[1]

DEST.dword[i] ← SIGNED\_DWORD\_SATURATE(DEST.dword[i] + p1dword + p2dword)

ELSE IF \*zeroing\*:

DEST.dword[i] ← 0

ELSE

DEST.dword[i] ← ORIGDEST.dword[i]

DEST[MAX\_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

```
VP4DPWSSDS __m512i _mm512_4dpwssds_epi32(__m512i, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_mask_4dpwssds_epi32(__m512i, __mmask16, __m512ix4, __m128i *);
VP4DPWSSDS __m512i _mm512_maskz_4dpwssds_epi32(__mmask16, __m512i, __m512ix4, __m128i *);
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4; additionally

- #UD If the EVEX broadcast bit is set to 1.
- #UD If the MODRM.mod = 0b11.



## VPERMB—Permute Packed Bytes Elements

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 8D /r VPERMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512VBMI	Permute bytes in xmm3/m128 using byte indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 8D /r VPERMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512VBMI	Permute bytes in ymm3/m256 using byte indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 8D /r VPERMB zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512VBMI	Permute bytes in zmm3/m512 using byte indexes in zmm2 and store the result in zmm1 using writemask k1.

### Instruction Operand Encoding

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

### Description

Copies bytes from the second source operand (the third operand) to the destination operand (the first operand) according to the byte indices in the first source operand (the second operand). Note that this instruction permits a byte in the source operand to be copied to more than one location in the destination operand.

Only the low 6(EVEX.512)/5(EVEX.256)/4(EVEX.128) bits of each byte index is used to select the location of the source byte from the second source operand.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated at byte granularity by the writemask k1.

### Operation

#### VPERMB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

n ← 3;

ELSE IF VL = 256:

n ← 4;

ELSE IF VL = 512:

n ← 5;

FI;

FOR j ← 0 TO KL-1:

id ← SRC1[j\*8 + n:j\*8]; // location of the source byte

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

DEST[j\*8 + 7:j\*8] ← SRC2[id\*8 + 7: id\*8];

ELSE IF zeroing-masking THEN

DEST[j\*8 + 7:j\*8] ← 0;

\*ELSE

DEST[j\*8 + 7:j\*8] remains unchanged\*

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

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

```
VPERMB __m512i __mm512_permutexvar_epi8(__m512i idx, __m512i a);  
VPERMB __m512i __mm512_mask_permutexvar_epi8(__m512i s, __mmask64 k, __m512i idx, __m512i a);  
VPERMB __m512i __mm512_maskz_permutexvar_epi8(__mmask64 k, __m512i idx, __m512i a);  
VPERMB __m256i __mm256_permutexvar_epi8(__m256i idx, __m256i a);  
VPERMB __m256i __mm256_mask_permutexvar_epi8(__m256i s, __mmask32 k, __m256i idx, __m256i a);  
VPERMB __m256i __mm256_maskz_permutexvar_epi8(__mmask32 k, __m256i idx, __m256i a);  
VPERMB __m128i __mm_permutexvar_epi8(__m128i idx, __m128i a);  
VPERMB __m128i __mm_mask_permutexvar_epi8(__m128i s, __mmask16 k, __m128i idx, __m128i a);  
VPERMB __m128i __mm_maskz_permutexvar_epi8(__mmask16 k, __m128i idx, __m128i a);
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.

## VPERMI2B—Full Permute of Bytes from Two Tables Overwriting the Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W0 75 /r VPERMI2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512VBMI	Permute bytes in xmm3/m128 and xmm2 using byte indexes in xmm1 and store the byte results in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 75 /r VPERMI2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512VBMI	Permute bytes in ymm3/m256 and ymm2 using byte indexes in ymm1 and store the byte results in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 75 /r VPERMI2B zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512VBMI	Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1.

### Instruction Operand Encoding

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

### Description

Permutes byte values in the second operand (the first source operand) and the third operand (the second source operand) using the byte indices in the first operand (the destination operand) to select byte elements from the second or third source operands. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result. The third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the same tables can be reused in subsequent iterations, but the index elements are overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****VPERMI2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

id ← 3;

ELSE IF VL = 256:

id ← 4;

ELSE IF VL = 512:

id ← 5;

FI;

TMP\_DEST[VL-1:0] ← DEST[VL-1:0];

FOR j ← 0 TO KL-1

off ← 8\*SRC1[j\*8 + id:j\*8];

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

DEST[j\*8 + 7:j\*8] ← TMP\_DEST[j\*8+id+1]? SRC2[off+7:off] : SRC1[off+7:off];

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7:j\*8] ← 0;

\*ELSE

DEST[j\*8 + 7:j\*8] remains unchanged\*

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

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

VPERMI2B \_\_m512i \_\_mm512\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI2B \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_mmask64 k, \_\_m512i b);

VPERMI2B \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMI2B \_\_m256i \_\_mm256\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2B \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_mmask32 k, \_\_m256i b);

VPERMI2B \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMI2B \_\_m128i \_\_mm\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMI2B \_\_m128i \_\_mm\_mask2\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_mmask16 k, \_\_m128i b);

VPERMI2B \_\_m128i \_\_mm\_maskz\_permutex2var\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.

## VPERMT2B—Full Permute of Bytes from Two Tables Overwriting a Table

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W0 7D /r VPERMT2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512VBMI	Permute bytes in xmm3/m128 and xmm1 using byte indexes in xmm2 and store the byte results in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 7D /r VPERMT2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512VBMI	Permute bytes in ymm3/m256 and ymm1 using byte indexes in ymm2 and store the byte results in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 7D /r VPERMT2B zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512VBMI	Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1.

### Instruction Operand Encoding

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

### Description

Permutes byte values from two tables, comprising of the first operand (also the destination operand) and the third operand (the second source operand). The second operand (the first source operand) provides byte indices to select byte results from the two tables. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result. The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the second table and the indices can be reused in subsequent iterations, but the first table is overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****VPERMT2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

id ← 3;

ELSE IF VL = 256:

id ← 4;

ELSE IF VL = 512:

id ← 5;

FI;

TMP\_DEST[VL-1:0] ← DEST[VL-1:0];

FOR j ← 0 TO KL-1

off ← 8\*SRC1[j\*8 + id:j\*8];

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

DEST[j\*8 + 7:j\*8] ← SRC1[j\*8+id+1]? SRC2[off+7:off] : TMP\_DEST[off+7:off];

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7:j\*8] ← 0;

\*ELSE

DEST[j\*8 + 7:j\*8] remains unchanged\*

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

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

VPERMT2B \_\_m512i\_mm512\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m512i\_mm512\_mask\_permutex2var\_epi8(\_\_m512i a, \_\_mmask64 k, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m512i\_mm512\_maskz\_permutex2var\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m256i\_mm256\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m256i\_mm256\_mask\_permutex2var\_epi8(\_\_m256i a, \_\_mmask32 k, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m256i\_mm256\_maskz\_permutex2var\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m128i\_mm\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMT2B \_\_m128i\_mm\_mask\_permutex2var\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m128i idx, \_\_m128i b);

VPERMT2B \_\_m128i\_mm\_maskz\_permutex2var\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.

## VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 7D /r VPERMT2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 7D /r VPERMT2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7D /r VPERMT2W zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW	Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 7E /r VPERMT2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 7E /r VPERMT2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 7E /r VPERMT2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W1 7E /r VPERMT2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 7E /r VPERMT2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7E /r VPERMT2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 7F /r VPERMT2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 7F /r VPERMT2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 7F /r VPERMT2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W1 7F /r VPERMT2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 7F /r VPERMT2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7F /r VPERMT2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

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

Description

Permutes 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table\_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table\_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMT2W (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

id ← 2

FI;

IF VL = 256

id ← 3

FI;

IF VL = 512

id ← 4

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1

i ← j \* 16

off ← 16\*SRC1[j+id:i]

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

THEN

DEST[i+15:i]=SRC1[j+id+1] ? SRC2[off+15:off]  
: TMP\_DEST[off+15:off]

ELSE

IF \*merging-masking\* ; merging-masking  
THEN \*DEST[i+15:i] remains unchanged\*  
ELSE ; zeroing-masking



```

        DEST[i+15:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### **VPERMT2D/VPERMT2PS (EVEX encoded versions)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id ← 1
FI;
IF VL = 256
    id ← 2
FI;
IF VL = 512
    id ← 3
FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1
    i ← j * 32
    off ← 32 * SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[31:0]
                    : TMP_DEST[off+31:off]
                ELSE
                    DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[off+31:off]
                    : TMP_DEST[off+31:off]
                FI
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
                FI
            FI
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

#### **VPERMT2Q/VPERMT2PD (EVEX encoded versions)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
    id ← 0
FI;
IF VL = 256
    id ← 1
FI;
IF VL = 512
    id ← 2
FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1

```

```

i ← j * 64
off ← 64 * SRC1[i+id:i]
IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[63:0]
                : TMP_DEST[off+63:off]
            ELSE
                DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[off+63:off]
                : TMP_DEST[off+63:off]
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

```

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

```

VPERMT2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMT2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMT2D __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMT2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMT2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMT2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMT2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMT2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMT2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMT2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMT2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMT2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMT2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMT2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);

```

```

VPERMT2PS __m256 __mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMT2PS __m256 __mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMT2PS __m256 __mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMT2PS __m256 __mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMT2PS __m128 __mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMT2PS __m128 __mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMT2PS __m128 __mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMT2PS __m128 __mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMT2Q __m512i __mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMT2Q __m512i __mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMT2Q __m512i __mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMT2Q __m512i __mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMT2Q __m256i __mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMT2Q __m256i __mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2Q __m256i __mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2Q __m256i __mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2Q __m128i __mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMT2Q __m128i __mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2Q __m128i __mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2Q __m128i __mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2W __m512i __mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMT2W __m512i __mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMT2W __m512i __mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMT2W __m512i __mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMT2W __m256i __mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMT2W __m256i __mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMT2W __m256i __mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERMT2W __m256i __mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMT2W __m128i __mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMT2W __m128i __mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2W __m128i __mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2W __m128i __mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

VPERMT2D/Q/PS/PD: See Exceptions Type E4NF.

VPERMT2W: See Exceptions Type E4NF.nb.

## VPMADD52LUQ—Packed Multiply of Unsigned 52-bit Integers and Add the Low 52-bit Products to Qword Accumulators

Opcode/ Instruction	Op/En	32/64 bit Mode Support	CPUID	Description
EVEX.DDS.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	AVX512IFMA AVX512VL	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 B4 /r VPMADD52LUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512IFMA AVX512VL	Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 B4 /r VPMADD52LUQ zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512IFMA	Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

### Instruction Operand Encoding

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

### Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

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

## Operation

### VPMADD52LUQ (EVEX encoded)

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

FOR j ← 0 TO KL-1

  i ← j \* 64;

  IF k1[j] OR \*no writemask\* THEN

    IF src2 is Memory AND EVEX.b=1 THEN

      tsrc2[63:0] ← ZeroExtend64(src2[51:0]);

    ELSE

      tsrc2[63:0] ← ZeroExtend64(src2[i+51:i]);

    FI;

    Temp128[127:0] ← ZeroExtend64(src1[i+51:i]) \* tsrc2[63:0];

    Temp2[63:0] ← DEST[i+63:i] + ZeroExtend64(temp128[51:0]);

    DEST[i+63:i] ← Temp2[63:0];

  ELSE

    IF \*zeroing-masking\* THEN

      DEST[i+63:i] ← 0;

    ELSE \*merge-masking\*

      DEST[i+63:i] is unchanged;

    FI;

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

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

VPMADD52LUQ \_\_m512i \_\_mm512\_madd52lo\_epu64( \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m512i \_\_mm512\_mask\_madd52lo\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m512i \_\_mm512\_maskz\_madd52lo\_epu64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_madd52lo\_epu64( \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_mask\_madd52lo\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_maskz\_madd52lo\_epu64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m128i \_\_mm\_madd52lo\_epu64( \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52LUQ \_\_m128i \_\_mm\_mask\_madd52lo\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52LUQ \_\_m128i \_\_mm\_maskz\_madd52lo\_epu64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E4.

## VPMADD52HUQ—Packed Multiply of Unsigned 52-bit Unsigned Integers and Add High 52-bit Products to 64-bit Accumulators

Opcode/ Instruction	Op/ En	32/64 bit Mode Support	CPUID	Description
EVEX.DDS.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512IFMA AVX512VL	Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512IFMA AVX512VL	Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512IFMA	Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

### Instruction Operand Encoding

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

### Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

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

**Operation****VPMADD52HUQ (EVEX encoded)**

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

FOR j ← 0 TO KL-1

i ← j \* 64;

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

IF src2 is Memory AND EVEX.b=1 THEN

tsrc2[63:0] ← ZeroExtend64(src2[51:0]);

ELSE

tsrc2[63:0] ← ZeroExtend64(src2[i+51:i]);

FI;

Temp128[127:0] ← ZeroExtend64(src1[i+51:i]) \* tsrc2[63:0];

Temp2[63:0] ← DEST[i+63:i] + ZeroExtend64(temp128[103:52]);

DEST[i+63:i] ← Temp2[63:0];

ELSE

IF \*zeroing-masking\* THEN

DEST[i+63:i] ← 0;

ELSE \*merge-masking\*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

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

VPMADD52HUQ \_\_m512i \_\_mm512\_madd52hi\_epu64(\_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m512i \_\_mm512\_mask\_madd52hi\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m512i \_\_mm512\_maskz\_madd52hi\_epu64(\_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_madd52hi\_epu64(\_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_mask\_madd52hi\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_maskz\_madd52hi\_epu64(\_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m128i \_\_mm\_madd52hi\_epu64(\_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52HUQ \_\_m128i \_\_mm\_mask\_madd52hi\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52HUQ \_\_m128i \_\_mm\_maskz\_madd52hi\_epu64(\_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPMULTISHIFTQB – Select Packed Unaligned Bytes from Quadword Sources

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 83 /r VPMULTISHIFTQB xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	AVX512VBMI AVX512VL	Select unaligned bytes from qwords in xmm3/m128/m64bcst using control bytes in xmm2, write byte results to xmm1 under k1.
EVEX.NDS.256.66.0F38.W1 83 /r VPMULTISHIFTQB ymm1 {k1}{z}, ymm2,ymm3/m256/m64bcst	A	V/V	AVX512VBMI AVX512VL	Select unaligned bytes from qwords in ymm3/m256/m64bcst using control bytes in ymm2, write byte results to ymm1 under k1.
EVEX.NDS.512.66.0F38.W1 83 /r VPMULTISHIFTQB zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512VBMI	Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1.

### Instruction Operand Encoding

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

### Description

This instruction selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

Only the low 6 bits of each control byte are used to select an 8-bit slot to extract the output byte from the qword data in the second source operand. The starting bit of the 8-bit slot can be unaligned relative to any byte boundary and is left-shifted from the beginning of the input qword source by the amount specified in the low 6-bit of the control byte. If the 8-bit slot would exceed the qword boundary, the out-of-bound portion of the 8-bit slot is wrapped back to start from bit 0 of the input qword element.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register.



**Operation****VPMULTISHIFTQB DEST, SRC1, SRC2 (EVEX encoded version)**

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

```

FOR i ← 0 TO KL-1
  IF EVEX.b=1 AND src2 is memory THEN
    tcur ← src2.qword[0]; //broadcasting
  ELSE
    tcur ← src2.qword[i];
  FI;
  FOR j ← 0 to 7
    ctrl ← src1.qword[i].byte[j] & 63;
    FOR k ← 0 to 7
      res.bit[k] ← tcur.bit[ (ctrl+k) mod 64 ];
    ENDFOR
    IF k1[i*8+j] or no writemask THEN
      dst.qword[i].byte[j] ← res;
    ELSE IF zeroing-masking THEN
      dst.qword[i].byte[j] ← 0;
    ENDFOR
  ENDFOR
ENDFOR
DEST.qword[MAX_VL-1:VL] ← 0;

```

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

```

VPMULTISHIFTQB __m512i __mm512_multishift_epi64_epi8( __m512i a, __m512i b);
VPMULTISHIFTQB __m512i __mm512_mask_multishift_epi64_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m512i __mm512_maskz_multishift_epi64_epi8( __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m256i __mm256_multishift_epi64_epi8( __m256i a, __m256i b);
VPMULTISHIFTQB __m256i __mm256_mask_multishift_epi64_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m256i __mm256_maskz_multishift_epi64_epi8( __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m128i __mm_multishift_epi64_epi8( __m128i a, __m128i b);
VPMULTISHIFTQB __m128i __mm_mask_multishift_epi64_epi8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULTISHIFTQB __m128i __mm_maskz_multishift_epi64_epi8( __mmask8 k, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.

## VPOPCNTD/VPOPCNTQ — Return the Count of Number of Bits Set to 1 in DWORD/QWORD

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 55 /r VPOPCNTD zmm1{k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512_VPOPCN TDQ	Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1.
EVEX.512.66.0F38.W1 55 /r VPOPCNTQ zmm1{k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512_VPOPCN TDQ	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	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

### Description

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

### Operation

#### VPOPCNTD

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

FOR j ← 0 TO KL-1

IF MaskBit(j) OR \*no writemask\*

DEST.dword[j] ← POPCNT(SRC.dword[j])

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\*

DEST.qword[j] ← POPCNT(SRC.qword[j])

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

VPOPCNTD \_\_m512i \_\_mm512\_popcnt\_epi32(\_\_m512i);

VPOPCNTD \_\_m512i \_\_mm512\_mask\_popcnt\_epi32(\_\_m512i, \_\_mmask16, \_\_m512i);

VPOPCNTD \_\_m512i \_\_mm512\_maskz\_popcnt\_epi32(\_\_mmask16, \_\_m512i);

VPOPCNTQ \_\_m512i \_\_mm512\_popcnt\_epi64(\_\_m512i);

VPOPCNTQ \_\_m512i \_\_mm512\_mask\_popcnt\_epi64(\_\_m512i, \_\_mmask8, \_\_m512i);

VPOPCNTQ \_\_m512i \_\_mm512\_maskz\_popcnt\_epi64(\_\_mmask8, \_\_m512i);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Type E4