5.1 TERNARY BIT VECTOR LOGIC TABLE

VPTERNLOGD/VPTERNLOGQ instructions operate on dword/qword elements and take three bit vectors of the respective input data elements to form a set of 32/64 indices, where each 3-bit value provides an index into an 8-bit lookup table represented by the imm8 byte of the instruction. The 256 possible values of the imm8 byte is constructed as a 16x16 boolean logic table. The 16 rows of the table uses the lower 4 bits of imm8 as row index. The 16 columns are referenced by imm8[7:4]. The 16 columns of the table are present in two halves, with 8 columns shown in Table 5-1 for the column index value between 0:7, followed by Table 5-2 showing the 8 columns corresponding to column index 8:15. This section presents the two-halves of the 256-entry table using a short-hand notation representing simple or compound boolean logic expressions with three input bit source data.

The three input bit source data will be denoted with the capital letters: A, B, C; where A represents a bit from the first source operand (also the destination operand), B and C represent a bit from the 2nd and 3rd source operands.

Each map entry takes the form of a logic expression consisting of one or more component expressions. Each component expression consists of either a unary or binary boolean operator and associated operands. Each binary boolean operator is expressed in lowercase letters, and operands concatenated after the logic operator. The unary operator 'not' is expressed using '!'. Additionally, the conditional expression "A?B:C" expresses a result returning B if A is set, returning C otherwise.

A binary boolean operator is followed by two operands, e.g. andAB. For a compound binary expression that contain commutative components and comprising the same logic operator, the 2nd logic operator is omitted and three operands can be concatenated in sequence, e.g. andABC. When the 2nd operand of the first binary boolean expression comes from the result of another boolean expression, the 2nd boolean expression is concatenated after the uppercase operand of the first logic expression, e.g. norBnandAC. When the result is independent of an operand, that operand is omitted in the logic expression, e.g. zeros or norCB.

The 3-input expression "majorABC" returns 0 if two or more input bits are 0, returns 1 if two or more input bits are 1. The 3-input expression "minorABC" returns 1 if two or more input bits are 0, returns 0 if two or more input bits are 1.

The building-block bit logic functions used in Table 5-1 and Table 5-2 include:

- Constants: TRUE (1), FALSE (0);
- Unary function: Not (!);
- Binary functions: and, nand, or, nor, xor, xnor;
- Conditional function: Select (?:);
- Tertiary functions: major, minor.
Table 5-1. Low 8 columns of the 16x16 Map of VTERNLOG Boolean Logic Operations

<table>
<thead>
<tr>
<th>Imm</th>
<th>08H</th>
<th>09H</th>
<th>0AH</th>
<th>0BH</th>
<th>0CH</th>
<th>0DH</th>
<th>0EH</th>
<th>0FH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0H</td>
<td>C?A:norBC</td>
<td>norAnorBC</td>
<td>xorAorBC</td>
<td>IA</td>
<td>andBIA</td>
<td>B?!A:norAC</td>
<td>B?!A:norAC</td>
<td>IA</td>
</tr>
<tr>
<td>1H</td>
<td>C?A:norAC</td>
<td>norBC</td>
<td>xorAC</td>
<td>IA</td>
<td>andBIA</td>
<td>B?!A:norAC</td>
<td>B?!A:norAC</td>
<td>IA</td>
</tr>
<tr>
<td>2H</td>
<td>C?A:norAC</td>
<td>norBA</td>
<td>xorAC</td>
<td>IA</td>
<td>andBIA</td>
<td>B?!A:norAC</td>
<td>B?!A:norAC</td>
<td>IA</td>
</tr>
<tr>
<td>3H</td>
<td>C?A:norAC</td>
<td>norBA</td>
<td>xorAC</td>
<td>IA</td>
<td>andBIA</td>
<td>B?!A:norAC</td>
<td>B?!A:norAC</td>
<td>IA</td>
</tr>
<tr>
<td>5H</td>
<td>C?A:norAC</td>
<td>norBA</td>
<td>xorAC</td>
<td>IA</td>
<td>andBIA</td>
<td>B?!A:norAC</td>
<td>B?!A:norAC</td>
<td>IA</td>
</tr>
<tr>
<td>6H</td>
<td>C?A:norAC</td>
<td>norBA</td>
<td>xorAC</td>
<td>IA</td>
<td>andBIA</td>
<td>B?!A:norAC</td>
<td>B?!A:norAC</td>
<td>IA</td>
</tr>
<tr>
<td>7H</td>
<td>C?A:norAC</td>
<td>norBA</td>
<td>xorAC</td>
<td>IA</td>
<td>andBIA</td>
<td>B?!A:norAC</td>
<td>B?!A:norAC</td>
<td>IA</td>
</tr>
</tbody>
</table>

Table 5-2 shows the half of 256-entry map corresponding to column index values 8:15.
Table 5-2. Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

<table>
<thead>
<tr>
<th>Imm</th>
<th>[7:4]</th>
<th>[3:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>08H</td>
<td>09H</td>
<td>0AH</td>
</tr>
<tr>
<td>andABC</td>
<td>andA\text{xor}BC</td>
<td>andCA</td>
</tr>
<tr>
<td>05H</td>
<td>A?andBC:IC</td>
<td>xnorCandBA</td>
</tr>
<tr>
<td>07H</td>
<td>xorA\text{xor}BC</td>
<td>A?\text{xnor}BC:and dB</td>
</tr>
<tr>
<td>08H</td>
<td>andCB</td>
<td>A?\text{xnor}BC:and dB</td>
</tr>
<tr>
<td>09H</td>
<td>B?C\text{nor}AC</td>
<td>xnorCB</td>
</tr>
<tr>
<td>0AH</td>
<td>A?andBBC</td>
<td>A?\text{xnor}BCC</td>
</tr>
<tr>
<td>0BH</td>
<td>B?C:IA</td>
<td>B?C\text{and}AC</td>
</tr>
<tr>
<td>0CH</td>
<td>A?andBCB</td>
<td>A?\text{xnor}BC:B</td>
</tr>
<tr>
<td>0DH</td>
<td>C?BIA</td>
<td>C?\text{BorAC}:B</td>
</tr>
<tr>
<td>0EH</td>
<td>A?andBCorBC</td>
<td>A?\text{xnor}BCorBC</td>
</tr>
<tr>
<td>0FH</td>
<td>nandA\text{and}B</td>
<td>nandA\text{xor}BC</td>
</tr>
</tbody>
</table>

Table 5-1 and Table 5-2 translate each of the possible value of the imm8 byte to a Boolean expression. These tables can also be used by software to translate Boolean expressions to numerical constants to form the imm8 value needed to construct the VPTERNLOG syntax. There is a unique set of three byte constants (F0H, CCH, AAH) that can be used for this purpose as input operands in conjunction with the Boolean expressions defined in those tables. The reverse mapping can be expressed as:

\[
\text{Result IMM8} = \text{Table Lookup Entry( F0H, 0CCH, 0AAH)}
\]

Table Lookup Entry is the Boolean expression defined in Table 5-1 and Table 5-2.
5.2 INSTRUCTIONS (V-Z)

VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

**Opcode/Instruction** | **Op / En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
EVEX.NDS.128.66.0F3A.W0 03 /r ib VALIGND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8 | FV | V/V | AVX512VL AVX512F | Shift right and merge vectors xmm2 and xmm3/m128/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.NDS.128.66.0F3A.W1 03 /r ib VALIGNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8 | FV | V/V | AVX512VL AVX512F | Shift right and merge vectors xmm2 and xmm3/m128/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W0 03 /r ib VALIGND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8 | FV | V/V | AVX512VL AVX512F | Shift right and merge vectors ymm2 and ymm3/m256/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.NDS.256.66.0F3A.W1 03 /r ib VALIGNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8 | FV | V/V | AVX512VL AVX512F | Shift right and merge vectors ymm2 and ymm3/m256/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W0 03 /r ib VALIGND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8 | FV | V/V | AVX512F | Shift right and merge vectors zmm2 and zmm3/m512/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.
EVEX.NDS.512.66.0F3A.W1 03 /r ib VALIGNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8 | FV | V/V | AVX512F | Shift right and merge vectors zmm2 and zmm3/m512/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand) into a 1024/512/256-bit intermediate vector. The low 512/256/128-bit of the intermediate vector is written to the destination operand (the first operand) using the writemask k1. The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values (merging-masking) or are set to 0 (zeroing-masking).
Operation

VALIGND (EVEX encoded versions)

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

IF \((\text{SRC2} \text{ is memory}) \text{ (AND EVEX}.b = 1)\)

\hspace{1em} \text{THEN}

\hspace{2em} \text{FOR } j \leftarrow 0 \text{ TO } KL-1

\hspace{3em} i \leftarrow j \ast 32

\hspace{3em} \text{src}[i+31:i] \leftarrow \text{SRC2}[31:0]

\hspace{2em} \text{ENDFOR;}

\hspace{2em} \text{ELSE src} \leftarrow \text{SRC2}

\hspace{1em} \text{FI}

\; \text{Concatenate sources}

\hspace{1em} \text{tmp}[VL-1:0] \leftarrow \text{src}[VL-1:0]

\hspace{1em} \text{tmp}[2VL-1:VL] \leftarrow \text{SRC1}[VL-1:0]

\; \text{Shift right doubleword elements}

\hspace{1em} \text{IF } VL = 128

\hspace{2em} \text{THEN } \text{SHIFT} = \text{imm8}[1:0]

\hspace{2em} \text{ELSE}

\hspace{3em} \text{IF } VL = 256

\hspace{4em} \text{THEN } \text{SHIFT} = \text{imm8}[2:0]

\hspace{4em} \text{ELSE } \text{SHIFT} = \text{imm8}[3:0]

\hspace{2em} \text{FI}

\hspace{1em} \text{FI}

\hspace{1em} \text{tmp}[2VL-1:0] \leftarrow \text{tmp}[2VL-1:0] \gg (32 \ast \text{SHIFT})

\; \text{Apply writemask}

\hspace{1em} \text{FOR } j \leftarrow 0 \text{ TO } KL-1

\hspace{2em} i \leftarrow j \ast 32

\hspace{2em} \text{IF } k1[j] \text{ OR } \text{no writemask*}

\hspace{3em} \text{THEN } \text{DEST}[i+31:i] \leftarrow \text{tmp}[i+31:i]

\hspace{2em} \text{ELSE}

\hspace{3em} \text{IF } \text{merging-masking*}

\hspace{4em} \text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged*}

\hspace{4em} \text{ELSE } \text{zeroing-masking}

\hspace{5em} \text{DEST}[i+31:i] \leftarrow 0

\hspace{2em} \text{FI}

\hspace{2em} \text{FI}

\text{ENDFOR;}

\hspace{1em} \text{DEST}[\text{MAX}_\text{VL}-1:VL] \leftarrow 0
VALIGNQ (EVEX encoded versions)

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

IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
        FOR j ← 0 TO KL-1
            i ← j * 64
            src[i+63:i] ← SRC2[63:0]
        ENDFOR;
        ELSE src ← SRC2
    FI

; Concatenate sources
tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]

; Shift right quadword elements
IF VL = 128
    THEN SHIFT = imm8[0]
ELSE
    IF VL = 256
        THEN SHIFT = imm8[1:0]
    ELSE SHIFT = imm8[2:0]
    FI
FI;
tmp[2VL-1:0] ← tmp[2VL-1:0] >> (64*SHIFT)

; Apply writemask
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← tmp[i+63:i]
    ELSE
        IF *merging-masking*
            THEN *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VALIGND __m512i _mm512_alignr_epi32( __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_mask_alignr_epi32( __m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_maskz_alignr_epi32( __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i _mm256_mask_alignr_epi32( __m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i _mm256_maskz_alignr_epi32( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i _mm_mask_alignr_epi32( __m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i _mm_maskz_alignr_epi32( __mmask8 k, __m128i a, __m128i b, int cnt);

Exceptions

See Exceptions Type E4NF.
VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

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

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source operand, 1 for second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 65 /r VBLENDMPD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend double-precision vector xmm2 and double-precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 65 /r VBLENDMPD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend double-precision vector ymm2 and double-precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 65 /r VBLENDMPD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend double-precision vector zmm2 and double-precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 65 /r VBLENDMPS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend single-precision vector xmm2 and single-precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 65 /r VBLENDMPS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend single-precision vector ymm2 and single-precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 65 /r VBLENDMPS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend single-precision vector zmm2 and single-precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.</td>
</tr>
</tbody>
</table>
Operation

**VBLENDMPD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no controlmask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+63:i] ← SRC2[63:0]
        ELSE
          DEST[i+63:i] ← SRC2[i+63:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN DEST[i+63:i] ← SRC1[i+63:i]
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0

**VBLENDMPS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no controlmask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] ← SRC2[31:0]
        ELSE
          DEST[i+31:i] ← SRC2[i+31:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN DEST[i+31:i] ← SRC1[i+31:i]
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI;
    FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VBLENDMPD __m512d __mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
VBLENDMPD __m256d __mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);
VBLENDMPD __m128d __mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);
VBLENDMPS __m512 __mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
VBLENDMPS __m256 __mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);
VBLENDMPS __m128 __mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.
## VBROADCAST—Load with Broadcast Floating-Point Data

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast single-precision floating-point element in mem to four locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, m32</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast single-precision floating-point element in mem to eight locations in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, m64</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast double-precision floating-point element in mem to four locations in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 1A /r VBROADCASTF128 ymm1, m128</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 19 /r VBROADCASTSD ymm1 {k1}{z}, xmm2/m64</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast low double-precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 19 /r VBROADCASTSD zmm1 {k1}{z}, xmm2/m64</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast low double-precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 19 /r VBROADCASTF32X2 ymm1 {k1}{z}, xmm2/m64</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Broadcast two single-precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 19 /r VBROADCASTF32X2 zmm1 {k1}{z}, xmm2/m64</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Broadcast two single-precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1 {k1}{z}, xmm2/m32</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast low single-precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1 {k1}{z}, xmm2/m32</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast low single-precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 18 /r VBROADCASTSS zmm1 {k1}{z}, xmm2/m32</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast low single-precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 1A /r VBROADCASTF32X4 ymm1 {k1}{z}, m128</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 1A /r VBROADCASTF32X4 zmm1 {k1}{z}, m128</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 1A /r VBROADCASTF64X2 ymm1 {k1}{z}, m128</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 1A /r VBROADCASTF64X2 zmm1 {k1}{z}, m128</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 1B /r VBROADCASTF64X8 zmm1 {k1}{z}, m256</td>
<td>T8</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Broadcast 256 bits of 8 single-precision floating-point data in mem to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 1B /r VBROADCASTF64X4 zmm1 {k1}{z}, m256</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast 256 bits of 4 double-precision floating-point data in mem to locations in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
Description
VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

VEX256-encoded versions: The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAX_VL-1:256) of the destination register are zeroed.

EVEX-encoded versions: The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

VBROADCASTSD and VBROADCASTF128,F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. VBROADCASTSS is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF64X4 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

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

If VBROADCAST or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Operation

**VBROADCASTSS (128 bit version VEX and legacy)**
- temp $\leftarrow$ SRC[31:0]
- DEST[31:0] $\leftarrow$ temp
- DEST[63:32] $\leftarrow$ temp
- DEST[95:64] $\leftarrow$ temp
- DEST[127:96] $\leftarrow$ temp
- DEST[MAX_VL-1:128] $\leftarrow$ 0
VBROADCASTSS (VEX.256 encoded version)

temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
DEST[MAX_VL-1:256] ← 0

VBROADCASTSS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
  FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VBROADCASTSD (VEX.256 encoded version)

temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAX_VL-1:256] ← 0

VBROADCASTSD (EVEX encoded versions)

(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← SRC[63:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
  FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VBROADCASTF32x2 (EVEX encoded versions)

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

FOR j ← 0 TO KL-1
    i ← j * 32
    n ← (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VBROADCASTF128 (VEX.256 encoded version)

temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAX_VL-1:256] ← 0

VBROADCASTF32X4 (EVEX encoded versions)

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

FOR j ← 0 TO KL-1
    i ← j * 32
    n ← (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VBROADCASTF64X2 (EVEX encoded versions)

\((KL, VL) = (4, 256), (8, 512)\)

FOR \(j \leftarrow 0\) TO \(KL-1\)
  \(i \leftarrow j \times 64\)
  \(n \leftarrow (j \mod 2) \times 64\)
  IF \(k1[j] \lor \) *no writemask*
    THEN \(DEST[i+63:i] \leftarrow SRC[n+63:n]\)
    ELSE
      IF *merging-masking* ; merging-masking
        THEN \(DEST[i+63:i] \) remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] = 0
      FI
  FI;
ENDFOR;

VBROADCASTF32X8 (EVEX.U1.512 encoded version)

FOR \(j \leftarrow 0\) TO 15
  \(i \leftarrow j \times 32\)
  \(n \leftarrow (j \mod 8) \times 32\)
  IF \(k1[j] \lor \) *no writemask*
    THEN \(DEST[i+31:i] \leftarrow SRC[n+31:n]\)
    ELSE
      IF *merging-masking* ; merging-masking
        THEN \(DEST[i+31:i] \) remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] = 0
      FI
  FI;
ENDFOR

\(DEST[MAX_VL-1:VL] \leftarrow 0\)

VBROADCASTF64X4 (EVEX.512 encoded version)

FOR \(j \leftarrow 0\) TO 7
  \(i \leftarrow j \times 64\)
  \(n \leftarrow (j \mod 4) \times 64\)
  IF \(k1[j] \lor \) *no writemask*
    THEN \(DEST[i+63:i] \leftarrow SRC[n+63:n]\)
    ELSE
      IF *merging-masking* ; merging-masking
        THEN \(DEST[i+63:i] \) remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] = 0
      FI
  FI;
ENDFOR

\(DEST[MAX_VL-1:VL] \leftarrow 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTF32x2 __m512 _mm512_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m512 _mm512_mask_broadcast_f32x2(__m128 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskz_broadcast_f32x2(__mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m512 _mm512_mask_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskz_broadcast_f32x4(__mmask16 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_broadcast_f32x8(__m256 a);
VBROADCASTF32x8 __m512 _mm512_mask_broadcast_f32x8(__m512 s, __mmask16 k, __m256 a);
VBROADCASTF32x8 __m512 _mm512_maskz_broadcast_f32x8(__mmask16 k, __m256 a);

VBROADCASTF64x2 __m512d _mm512d_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m512d _mm512d_mask_broadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d _mm512d_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256d_broadcast_f64x2(__m256d a);
VBROADCASTF64x2 __m256d _mm256d_mask_broadcast_f64x2(__m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256d_maskz_broadcast_f64x2(__mmask8 k, __m128d a);

VBROADCASTSD __m512d _mm512d_broadcastsd_pd(__m128d a);
VBROADCASTSD __m512d _mm512d_mask_broadcastsd_pd(__m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512d_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256d_broadcastsd_pd(__m256d a);
VBROADCASTSD __m256d _mm256d_mask_broadcastsd_pd(__m256d s, __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256d_maskz_broadcastsd_pd(__mmask8 k, __m128d a);

VBROADCASTSS __m512 _mm512_broadcastss_ps(__m128 a);
VBroadcastSS __m512 _mm512_mask_broadcastss_ps(__m128 s, __mmask16 k, __m128 a);
VBroadcastSS __m512 _mm512_maskz_broadcastss_ps(__mmask16 k, __m128 a);
VBroadcastSS __m256d _mm256d_broadcastss_ps(__m256d a);
VBroadcastSS __m256d _mm256d_mask_broadcastss_ps(__m256d s, __mmask8 k, __m128d a);
VBroadcastSS __m256d _mm256d_maskz_broadcastss_ps(__mmask8 k, __m128d a);

Exceptions

VEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, see Exceptions Type E6.

#UD
  If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.
  If EVEX.L’L = 0 for VBROADCASTSD/VBROADCASF32X2/VBROADCASF32X4/VBROADCASF64X2.
  If EVEX.L’L < 10b for VBROADCASF32X8/VBROADCASF64X4.
VPBROADCASTM—Broadcast Mask to Vector Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W1 2A /r VPBROADCASTMB2Q xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low byte value in k1 to two locations in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 2A /r VPBROADCASTMB2Q ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low byte value in k1 to four locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 2A /r VPBROADCASTMB2Q zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Broadcast low byte value in k1 to eight locations in zmm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 3A /r VPBROADCASTMW2D xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low word value in k1 to four locations in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 3A /r VPBROADCASTMW2D ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low word value in k1 to eight locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 3A /r VPBROADCASTMW2D zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Broadcast low word value in k1 to sixteen locations in zmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128). EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

**VPBROADCASTMB2Q**

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

FOR j ← 0 TO KL-1

i ← j*64

DEST[i+63:j] ← ZeroExtend(SRC[7:0])

ENDFOR

DEST[MAX_VL-1:VL] ← 0

**VPBROADCASTMW2D**

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

FOR j ← 0 TO KL-1

i ← j*32

DEST[i+31:j] ← ZeroExtend(SRC[15:0])

ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q __m512i __mm512_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m512i __mm512_broadcastmw_epi32(__mmask16);
VPBROADCASTMB2Q __m256i __mm256_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m256i __mm256_broadcastmw_epi32(__mmask8);
VPBROADCASTMB2Q __m128i __mm_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m128i __mm_broadcastmw_epi32(__mmask8);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E6NF.
VCOMPRESSPD—Store Sparse Packed Double-Precision Floating-Point Values into Dense Memory

Table: VCOMPRESSPD—Instruction Set Reference

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 8A /r VCOMPRESSPD xmm1/m128 (k1)[z], xmm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed double-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 8A /r VCOMPRESSPD ymm1/m256 (k1)[z], ymm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed double-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 8A /r VCOMPRESSPD zmm1/m512 (k1)[z], zmm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed double-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.</td>
</tr>
</tbody>
</table>

### InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Compress (store) up to 8 double-precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

**VCOMPRESSPD (EVEX encoded versions) store form**

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

\[\text{SIZE} \leftarrow 64\]

\[k \leftarrow 0\]

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\[i \leftarrow j \times 64\]

IF k1[j] OR "no writemask"

THEN

\[\text{DEST}[k+\text{SIZE}-1:k] \leftarrow \text{SRC}[i+63:i]\]

\[k \leftarrow k + \text{SIZE}\]

FI;

ENDFOR
VCOMPRESSPD (EVEX encoded versions) reg-reg form

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

SIZE ← 64
k ← 0
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
  THEN
    DEST[k+SIZE-1:k] ← SRC[i+63:i]
    k ← k + SIZE
  FI;
ENDFOR
IF *merging-masking*
  THEN *DEST[VL-1:k] remains unchanged*
  ELSE DEST[VL-1:k] ← 0
FI
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCOMPRESSPD __m512d __mm512_mask_compress_pd(__m512d s, __mmask8 k, __m512d a);
VCOMPRESSPD __m512d __mm512_maskz_compress_pd(__mmask8 k, __m512d a);
VCOMPRESSPD void __nm512_mask_compressstoreu_pd(void * d, __mmask8 k, __m512d a);
VCOMPRESSPD __m256d __mm256_mask_compress_pd(__m256d s, __mmask8 k, __m256d a);
VCOMPRESSPD __m256d __mm256_maskz_compress_pd(__mmask8 k, __m256d a);
VCOMPRESSPD void __nm256_mask_compressstoreu_pd(void * d, __mmask8 k, __m256d a);
VCOMPRESSPD __m128d __mm128_mask_compress_pd(__m128d s, __mmask8 k, __m128d a);
VCOMPRESSPD __m128d __mm128_maskz_compress_pd(__mmask8 k, __m128d a);
VCOMPRESSPD void __nm128_mask_compressstoreu_pd(void * d, __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.
VCOMPRESSPS—Store Sparse Packed Single-Precision Floating-Point Values into Dense Memory

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 8A /r VCOMPRESSPS xmm1/m128 (k1)[z], xmm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed single-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8A /r VCOMPRESSPS ymm1/m256 (k1)[z], ymm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed single-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8A /r VCOMPRESSPS zmm1/m512 (k1)[z], zmm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed single-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM/tr/m (w)</td>
<td>ModRM/reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compress (stores) up to 16 single-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register; the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (a partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

VCOMPRESSPS (EVEX encoded versions) store form

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

\[ \text{SIZE} \leftarrow 32 \]

\[ k \leftarrow 0 \]

FOR \( j \leftarrow 0 \) TO KL-1

\[ i \leftarrow j \times 32 \]

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

THEN

\[ \text{DEST}[k+\text{SIZE}-1:k] \leftarrow \text{SRC}[i+31:i] \]

\[ k \leftarrow k + \text{SIZE} \]

FI;

ENDFOR;
**VCOMPRESSPS (EVEX encoded versions) reg-reg form**

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

SIZE ← 32

k ← 0

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      DEST[k+SIZE-1:k] ← SRC[i+31:i]
      k ← k + SIZE
    FI;
  ENDFOR

IF *merging-masking*
  THEN *DEST[VL-1:k] remains unchanged*
  ELSE DEST[VL-1:k] ← 0
  FI

DEST[MAX_VL-1:VL] ← 0

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

VCOMPRESSPS __m512 _mm512_mask_compress_ps( __m512 s, __mmask16 k, __m512 a);
VCOMPRESSPS __m512 __mm512_mask2_compress_ps( __mmask16 k, __m512 a);
VCOMPRESSPS void _mm512_mask_compressstoreu_ps( void * d, __mmask16 k, __m512 a);
VCOMPRESSPS __m256 _mm256_mask_compress_ps( __m256 s, __mmask8 k, __m256 a);
VCOMPRESSPS __m256 _mm256_maskz_compress_ps( __mmask8 k, __m256 a);
VCOMPRESSPS void _mm256_mask_compressstoreu_ps( void * d, __mmask8 k, __m256 a);
VCOMPRESSPS __m128 _mm_mask_compress_ps( __m128 s, __mmask8 k, __m128 a);
VCOMPRESSPS __m128 _mm_maskz_compress_ps( __mmask8 k, __m128 a);
VCOMPRESSPS void _mm_mask_compressstoreu_ps( void * d, __mmask8 k, __m128 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.
VCVTPD2QQ—Convert Packed Double-Precision Floating-Point Values to Packed Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1 7B /r VCVTPD2QQ xmm1 (k1)[z], xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed quadword integers in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 7B /r VCVTPD2QQ ymm1 (k1)[z], ymm2/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 7B /r VCVTPD2QQ zmm1 (k1)[z], zmm2/m512/m64bcst(ER)</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double-precision floating-point values from zmm2/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^{w-1}\), where \(w\) represents the number of bits in the destination format) is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VCVTPD2QQ (EVEX encoded version) when src operand is a register**

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

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

THEN

    SET_RM(EVEX.RC);

ELSE

    SET_RM(MXCSR.RM);

Fi;

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])

ELSE

    IF *merging-masking* ; merging-masking

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

    ELSE ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX_VL-1:VL] ← 0

**VCVTPD2QQ (EVEX encoded version) when src operand is a memory source**

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

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN

    IF (EVEX.b == 1)

        THEN

            DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[63:0])

        ELSE

            DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])

    FI;

ELSE

    IF *merging-masking* ; merging-masking

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

    ELSE ; zeroing-masking

        DEST[i+63:i] ← 0

    FI

FI;

ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2QQ _m512i _mm512_cvtpd_epi64( __m512d a);
VCVTPD2QQ _m512i _mm512_mask_cvtpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ _m512i _mm512_maskz_cvtpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ _m512i _mm512_cvt_roundpd_epi64( __m512d a, int r);
VCVTPD2QQ _m512i _mm512_mask_cvt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ _m512i _mm512_maskz_cvt_roundpd_epi64( __mmask8 k, __m512d a, int r);
VCVTPD2QQ _m256i _mm256_mask_cvtpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ _m256i _mm256_maskz_cvtpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ _m128i _mm_mask_cvtpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2QQ _m128i _mm_maskz_cvtpd_epi64( __mmask8 k, __m128d a);
VCVTPD2QQ _m256i _mm256_cvtpd_epi64( __m256d src);
VCVTPD2QQ _m128i _mm_cvtpd_epi64( __m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.
VCVTPD2UDQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W1 79 /r VCVTPD2UDQ xmm1 [k1]{z}, xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W1 79 /r VCVTPD2UDQ xmm1 [k1]{z}, ymm2/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W1 79 /r VCVTPD2UDQ ymm1 [k1]{z}, zmm2/m512/m64bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAX_VL-1:256) of the corresponding destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VCVTPD2UDQ (EVEX encoded versions) when src2 operand is a register**

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

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

THEN

\[\text{SET}_R(MXCSR.RM);\]

ELSE

\[\text{SET}_R(MXCSR.RM);\]

FI;

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\[i \leftarrow j \times 32\]

\[k \leftarrow j \times 64\]

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

THEN

\[\text{DEST}[i+31:j] \leftarrow \text{Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger}(\text{SRC}[k+63:k])\]

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

\[\text{DEST}[i+31:j] \leftarrow 0\]

FI

FI;

ENDFOR

\[\text{DEST}[\text{MAX\_VL-1:VL/2}] \leftarrow 0\]

**VCVTPD2UDQ (EVEX encoded versions) when src operand is a memory source**

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

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\[i \leftarrow j \times 32\]

\[k \leftarrow j \times 64\]

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

THEN

IF \((EVEX.b = 1)\)

THEN

\[\text{DEST}[i+31:j] \leftarrow \text{Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger}(\text{SRC}[63:0])\]

ELSE

\[\text{DEST}[i+31:j] \leftarrow \text{Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger}(\text{SRC}[k+63:k])\]

FI;

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

\[\text{DEST}[i+31:j] \leftarrow 0\]

FI

FI;

ENDFOR

\[\text{DEST}[\text{MAX\_VL-1:VL/2}] \leftarrow 0\]
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPD2UDQ __m256i _mm512_cvtpd_epu32( __m512d a);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_maskz_cvtpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_cvt_roundpd_epu32( __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_mask_cvt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i _mm256_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm256_maskz_cvtpd_epu32( __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i _mm_maskz_cvtpd_epu32( __mmask8 k, __m128d a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTPD2UQQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1 79 /r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double-precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 79 /r VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert fourth packed double-precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 79 /r VCVTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst(er)</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

VCVTDP2UQQ (EVEX encoded versions) when src operand is a register

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

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

\[\text{THEN} \]

\[\text{SET} \_ \text{RM}(EVEX.RC);\]

\[\text{ELSE}\]

\[\text{SET} \_ \text{RM}(MXCSR.RM);\]

\[\text{FI};\]

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\[i \leftarrow j * 64\]

IF \(k1[j]\) OR \(*\text{no writemask}*\)

\[\text{THEN} \quad \text{DEST}[i+63:i] \leftarrow\]

\[\text{Convert} \_ \text{Double} \_ \text{Precision} \_ \text{Floating} \_ \text{Point} \_ \text{To} \_ \text{UQuadInteger}(\text{SRC}[i+63:i])\]

\[\text{ELSE}\]

IF \(*\text{merging-masking}*\) ; merging-masking

\[\text{THEN} \quad \text{DEST}[i+63:i] \quad \text{remains unchanged}^*\]

\[\text{ELSE}\]

\[\text{zeroing-masking}\]

\[\text{DEST}[i+63:i] \leftarrow 0\]

\[\text{FI}\]

\[\text{FI};\]

ENDFOR

\[\text{DEST}[\text{MAX}_\_\text{VL}-1:\text{VL}] \leftarrow 0\]

VCVTDP2UQQ (EVEX encoded versions) when src operand is a memory source

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

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\[i \leftarrow j * 64\]

IF \(k1[j]\) OR \(*\text{no writemask}*\)

\[\text{THEN}\]

IF \((EVEX.b == 1)\)

\[\text{THEN}\]

\[\text{DEST}[i+63:i] \leftarrow\]

\[\text{Convert} \_ \text{Double} \_ \text{Precision} \_ \text{Floating} \_ \text{Point} \_ \text{To} \_ \text{UQuadInteger}(\text{SRC}[63:0])\]

\[\text{ELSE}\]

\[\text{DEST}[i+63:i] \leftarrow\]

\[\text{Convert} \_ \text{Double} \_ \text{Precision} \_ \text{Floating} \_ \text{Point} \_ \text{To} \_ \text{UQuadInteger}(\text{SRC}[i+63:i])\]

\[\text{FI};\]

ELSE

IF \(*\text{merging-masking}*\) ; merging-masking

\[\text{THEN} \quad \text{DEST}[i+63:i] \quad \text{remains unchanged}^*\]

\[\text{ELSE}\]

\[\text{zeroing-masking}\]

\[\text{DEST}[i+63:i] \leftarrow 0\]

\[\text{FI}\]

\[\text{FI};\]

ENDFOR

\[\text{DEST}[\text{MAX}_\_\text{VL}-1:\text{VL}] \leftarrow 0\]
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2UQQ __m512i _mm512_cvtpd_epu64(__m512d a);
VCVTPD2UQQ __m512i _mm512_mask_cvtpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_mask_z_cvtpd_epu64(__mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_cvt_roundpd_epu64(__m512d a, int r);
VCVTPD2UQQ __m512i _mm512_mask_cvt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_maskz_cvt_roundpd_epu64(__mmask8 k, __m512d a);
VCVTPD2UQQ __m256i _mm256_mask_cvtpd_epu64(__m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i _mm256_maskz_cvtpd_epu64(__mmask8 k, __m256d a);
VCVTPD2UQQ __m128i _mm_mask_cvtpd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i _mm_maskz_cvtpd_epu64(__mmask8 k, __m128d a);
VCVTPD2UQQ __m256i _mm256_cvt_roundpd_epu64(__m256d src);
VCVTPD2UQQ __m128i _mm_cvt_roundpd_epu64(__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD If EVEX.vvvv != 1111B.
VCVTPH2PS—Convert 16-bit FP values to Single-Precision FP values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.6F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64</td>
<td>RM V/V</td>
<td>F16C</td>
<td>Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.6F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128</td>
<td>RM V/V</td>
<td>F16C</td>
<td>Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.6F38.W0 13 /r VCVTPH2PS xmm1 {k1}[z], xmm2/m64</td>
<td>HVM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.6F38.W0 13 /r VCVTPH2PS ymm1 {k1}[z], xmm2/m128</td>
<td>HVM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point values in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.6F38.W0 13 /r VCVTPH2PS zmm1 {k1}[z], ymm2/m256 {sae}</td>
<td>HVM V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed half precision (16-bit) floating-point values in ymm2/m256 to packed single-precision floating-point values in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>HVM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed half precision (16-bits) floating-point values in the low-order bits of the source operand (the second operand) to packed single-precision floating-point values and writes the converted values into the destination operand (the first operand).

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

VEX.256 version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAX_VL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) FP values.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).
**Operation**

\[
v\text{Cvt\_h2s}(\text{SRC1}[15:0])
\]

\[
\{
\text{RETURN Cvt\_Half\_Precision\_To\_Single\_Precision(SRC1[15:0])};
\}
\]

**VCVTPH2PS (EVEX encoded versions)**

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

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\[
i \leftarrow j \times 32
\]

\[
k \leftarrow j \times 16
\]

IF \( k1[j] \) OR *no writemask*  
THEN \( \text{DEST}[i+31:i] \leftarrow \text{vCvt\_h2s} (\text{SRC}[k+15:k]) \)
ELSE

IF *merging-masking*  
THEN *DEST[i+31:i] remains unchanged*  
ELSE  

\[
\text{DEST}[i+31:i] \leftarrow 0
\]

FI

FI;

ENDFOR

\( \text{DEST}[\text{MAX\_VL-1:VL}] \leftarrow 0 \)

**VCVTPH2PS (VEX.256 encoded version)**

\( \text{DEST}[31:0] \leftarrow \text{vCvt\_h2s} (\text{SRC1}[15:0]) \);

\( \text{DEST}[63:32] \leftarrow \text{vCvt\_h2s} (\text{SRC1}[31:16]) \);

\( \text{DEST}[95:64] \leftarrow \text{vCvt\_h2s} (\text{SRC1}[47:32]) \);

\( \text{DEST}[127:96] \leftarrow \text{vCvt\_h2s} (\text{SRC1}[63:48]) \);

\( \text{DEST}[159:128] \leftarrow \text{vCvt\_h2s} (\text{SRC1}[79:64]) \);

\( \text{DEST}[191:160] \leftarrow \text{vCvt\_h2s} (\text{SRC1}[95:80]) \);

\( \text{DEST}[223:192] \leftarrow \text{vCvt\_h2s} (\text{SRC1}[111:96]) \);

\( \text{DEST}[255:224] \leftarrow \text{vCvt\_h2s} (\text{SRC1}[127:112]) \);

\( \text{DEST}[\text{MAX\_VL-1:256}] \leftarrow 0 \)
VCVTPH2PS (VEX.128 encoded version)
DEST[31:0] ← vCvt_h2s(SRC1[15:0]);
DEST[63:32] ← vCvt_h2s(SRC1[31:16]);
DEST[95:64] ← vCvt_h2s(SRC1[47:32]);
DEST[127:96] ← vCvt_h2s(SRC1[63:48]);
DEST[MAX_VL-1:128] ← 0

Flags Affected
None

Intel C/C++ Compiler Intrinsic Equivalent
VCVTPH2PS __m512 __m512_cvtph_ps(__m256i a);
VCVTPH2PS __m512 __m512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);
VCVTPH2PS __m512 __m512_maskz_cvtph_ps(__mmask16 k, __m256i a);
VCVTPH2PS __m512 __m512_cvt_roundph_ps(__m256i a, int sae);
VCVTPH2PS __m512 __m512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m512 __m512_maskz_cvt_roundph_ps(__mmask16 k, __m256i a, int sae);
VCVTPH2PS __m256 __m256_mask_cvtph_ps(__m256 s, __mmask8 k, __m128i a);
VCVTPH2PS __m256 __m256_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 __m128_mask_cvtph_ps(__m128 s, __mmask8 k, __m128i a);
VCVTPH2PS __m128 __m128_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 __m128_cvtph_ps (__m128i m1);
VCVTPH2PS __m256 __m256_cvtph_ps (__m128i m1)

SIMD Floating-Point Exceptions
Invalid

Other Exceptions
VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);
EVEX-encoded instructions, see Exceptions Type E11.

#UD If VEX.W=1.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F3A,W0 1D / r ib</td>
<td>MRI</td>
<td>V/V</td>
<td>F16C</td>
<td>Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A,W0 1D / r ib</td>
<td>MRI</td>
<td>V/V</td>
<td>F16C</td>
<td>Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A,W0 1D / r ib</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A,W0 1D / r ib</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A,W0 1D / r ib</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>HVM</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Convert packed single-precision floating values in the source operand to half-precision (16-bit) floating-point values and store to the destination operand. The rounding mode is specified using the immediate field (imm8). Underflow results (i.e., tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to the input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 5-3.

Figure 5-7. VCVTPS2PH (128-bit Version)
VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value

INSTRUCTION SET REFERENCE, V-Z

VEX.128 version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If the destination operand is a register then the upper bits (MAX_VL-1:64) of corresponding register are zeroed.

VEX.256 version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAX_VL-1:256/128/64) of the corresponding destination register are zeroed.

Operation

```
vCvt_s2h(SRC1[31:0])
{
  IF Imm[2] = 0
  THEN ; using Imm[1:0] for rounding control, see Table 5-3
     RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
  ELSE ; using MXCSR.RC for rounding
     RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
  FI;
}
```

Table 5-3. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field Name/value</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm[1:0]</td>
<td></td>
<td>Round to nearest even</td>
<td>If Imm[2] = 0</td>
</tr>
<tr>
<td></td>
<td>RC=00B</td>
<td>Round down</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RC=01B</td>
<td>Round up</td>
<td></td>
</tr>
<tr>
<td>Imm[2]</td>
<td>MS1=0</td>
<td>Use imm[1:0] for rounding</td>
<td>Ignore MXCSR.RC</td>
</tr>
<tr>
<td></td>
<td>MS1=1</td>
<td>Use MXCSR.RC for rounding</td>
<td></td>
</tr>
<tr>
<td>Imm[7:3]</td>
<td></td>
<td>Ignored</td>
<td>Ignored by processor</td>
</tr>
</tbody>
</table>

VCVTPS2PH (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:j] ← vCvt_s2h(SRC[k+31:k])
    ELSE ; merging-masking
      IF *merging-masking*
        THEN DEST[i+15:j] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+15:j] ← 0
      FI
    ENDIF;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0
```
VCVTPS2PH (EVEX encoded versions) when dest is memory

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

FOR \(j \leftarrow 0\) TO \(KL-1\)

\[i \leftarrow j \times 16\]

\[k \leftarrow j \times 32\]

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

THEN \(\text{DEST}[i+15:i] \leftarrow \text{vCvt_s2h(SRC}[k+31:k])\)

ELSE

*\(\text{DEST}[i+15:i]\) remains unchanged* ; merging-masking

\FI;

ENDFOR

VCVTPS2PH (VEX.256 encoded version)

\(\text{DEST}[15:0] \leftarrow \text{vCvt_s2h(SRC}[1:31]);\)

\(\text{DEST}[31:16] \leftarrow \text{vCvt_s2h(SRC}[32:63]);\)

\(\text{DEST}[47:32] \leftarrow \text{vCvt_s2h(SRC}[64:95]);\)

\(\text{DEST}[63:48] \leftarrow \text{vCvt_s2h(SRC}[96:127]);\)

\(\text{DEST}[79:64] \leftarrow \text{vCvt_s2h(SRC}[128:159]);\)

\(\text{DEST}[111:96] \leftarrow \text{vCvt_s2h(SRC}[160:191]);\)

\(\text{DEST}[127:112] \leftarrow \text{vCvt_s2h(SRC}[192:223]);\)

\(\text{DEST}[\text{MAX}_V\_L-1:128] \leftarrow 0\)

VCVTPS2PH (VEX.128 encoded version)

\(\text{DEST}[15:0] \leftarrow \text{vCvt_s2h(SRC}[1:31]);\)

\(\text{DEST}[31:16] \leftarrow \text{vCvt_s2h(SRC}[32:63]);\)

\(\text{DEST}[47:32] \leftarrow \text{vCvt_s2h(SRC}[64:95]);\)

\(\text{DEST}[63:48] \leftarrow \text{vCvt_s2h(SRC}[96:127]);\)

\(\text{DEST}[\text{MAX}_V\_L-1:64] \leftarrow 0\)

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{VCVTPS2PH} \_\text{m256i} \_\text{mm512} \_\text{cvtps}\_\text{ph}\(_\text{m512} \_\text{a});
\text{VCVTPS2PH} \_\text{m256i} \_\text{mm512} \_\text{mask}\_\text{cvtps}\_\text{ph}\(_\text{m512} \_\text{a});
\text{VCVTPS2PH} \_\text{m256i} \_\text{mm512} \_\text{cvt_roundps}\_\text{ph}\(_\text{m512} \_\text{a, const int imm});
\text{VCVTPS2PH} \_\text{m256i} \_\text{mm512} \_\text{mask}\_\text{cvt_roundps}\_\text{ph}\(_\text{m512} \_\text{a, const int imm});
\text{VCVTPS2PH} \_\text{m128i} \_\text{mm256} \_\text{cvtps}\_\text{ph}\(_\text{m128} \_\text{a});
\text{VCVTPS2PH} \_\text{m128i} \_\text{mm256} \_\text{mask}\_\text{cvtps}\_\text{ph}\(_\text{m128} \_\text{a});
\text{VCVTPS2PH} \_\text{m128i} \_\text{mm256} \_\text{cvt_roundps}\_\text{ph}\(_\text{m128} \_\text{a, const int imm});
\text{VCVTPS2PH} \_\text{m128i} \_\text{mm256} \_\text{mask}\_\text{cvt_roundps}\_\text{ph}\(_\text{m128} \_\text{a});
\end{align*}
\]

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);
Other Exceptions
VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);
EVEX-encoded instructions, see Exceptions Type E11.
#UD If VEX.W=1.
#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VCVTPS2UDQ—Convert Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W0 79 /r VCVTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 79 /r VCVTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 79 /r VCVTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts sixteen packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

VCVTPS2UDQ (EVEX encoded versions) when src operand is a register

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

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

THEN

    SET_RM(EVEX.RC);

ELSE

    SET_RM(MXCSR.RM);

FI;

FOR j \(\leftarrow\) 0 TO KL-1

i \(\leftarrow\) j * 32

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] \(\leftarrow\)

Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])

ELSE

    IF *merging-masking* ; merging-masking

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

    ELSE ; zeroing-masking

        DEST[i+31:i] \(\leftarrow\) 0

    FI

FI;

ENDFOR

DEST[MAX_VL-1:VL] \(\leftarrow\) 0

VCVTPS2UDQ (EVEX encoded versions) when src operand is a memory source

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

FOR j \(\leftarrow\) 0 TO KL-1

i \(\leftarrow\) j * 32

IF k1[j] OR *no writemask*

THEN

    IF (EVEX.b = 1)

    THEN

        DEST[i+31:i] \(\leftarrow\)

        Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0])

    ELSE

        DEST[i+31:i] \(\leftarrow\)

        Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])

    FI;

ELSE

    IF *merging-masking* ; merging-masking

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

    ELSE ; zeroing-masking

        DEST[i+31:i] \(\leftarrow\) 0

    FI

FI;

ENDFOR

DEST[MAX_VL-1:VL] \(\leftarrow\) 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPS2UDQ __m512i _mm512_cvtps_epu32( __m512 a);
VCVTPS2UDQ __m512i _mm512_mask_cvtps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_maskz_cvtps_epu32( __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_cvt_roundps_epu32( __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_mask_cvt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_maskz_cvt_roundps_epu32( __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i _mm256_cvtps_epu32( __m256d a);
VCVTPS2UDQ __m256i _mm256_mask_cvtps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i _mm256_maskz_cvtps_epu32( __mmask8 k, __m256 a);
VCVTPS2UDQ __m128i _mm_cvtps_epu32( __m128 a);
VCVTPS2UDQ __m128i _mm_mask_cvtps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i _mm_maskz_cvtps_epu32( __mmask8 k, __m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 7B /r VCVTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 7B /r VCVTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 7B /r VCVTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts eight packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^{w-1}\), where \(w\) represents the number of bits in the destination format) is returned.

The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

**VCVTPS2QQ (EVEX encoded versions) when src operand is a register**

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

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

THEN

    SET_RM(EVEX.RC);

ELSE

    SET_RM(MXCSR.RM);

FI;

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\(i \leftarrow j \times 64\)

\(k \leftarrow j \times 32\)

IF \(k1[\(j\)] OR \"no writemask\"

THEN \(DEST[i+63:i] \leftarrow\)

Convert_Single_Precision_To_QuadInteger(SRC\([k+31:k]\))

ELSE

IF \"merging-masking\"

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

ELSE \"zeroing-masking\"

\(DEST[i+63:i] \leftarrow 0\)

FI

FI;

ENDFOR

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

**VCVTPS2QQ (EVEX encoded versions) when src operand is a memory source**

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

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\(i \leftarrow j \times 64\)

\(k \leftarrow j \times 32\)

IF \(k1[\(j\)] OR \"no writemask\"

THEN

IF \((EVEX.b == 1)\)

THEN

\(DEST[i+63:i] \leftarrow\)

Convert_Single_Precision_To_QuadInteger(SRC\([31:0]\))

ELSE

\(DEST[i+63:i] \leftarrow\)

Convert_Single_Precision_To_QuadInteger(SRC\([k+31:k]\))

FI;

ELSE

IF \"merging-masking\"

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

ELSE \"zeroing-masking\"

\(DEST[i+63:i] \leftarrow 0\)

FI

FI;

ENDFOR

\(DEST[MAX\_VL-1:VL] \leftarrow 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2QQ __m512i _mm512_cvtps_epi64( __m512 a);
VCVTPS2QQ __m512i _mm512_mask_cvtps_epi64( __m512 s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_cvt_roundps_epi64( __m512 a, int r);
VCVTPS2QQ __m512i _mm512_mask_cvt_roundps_epi64( __m512 s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i _mm512_maskz_cvt_roundps_epi64( __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i _mm256_cvtps_epi64( __m256 a);
VCVTPS2QQ __m256i _mm256_mask_cvtps_epi64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i _mm256_maskz_cvtps_epi64( __mmask8 k, __m256 a);
VCVTPS2QQ __m128i _mm128_cvtps_epi64( __m128 a);
VCVTPS2QQ __m128i _mm128_mask_cvtps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i _mm128_maskz_cvtps_epi64( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E3
#UD If EVEX.vvvv != 1111B.
VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op /En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 79 /r VCVTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 79 /r VCVTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

**VCVTPS2UQQ (EVEX encoded versions) when src operand is a register**

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

If \((VL == 512) AND (EVEX.b == 1)\)

THEN

\[
\text{SET\_RM(EVEX.RC)};
\]

ELSE

\[
\text{SET\_RM(MXCSR.RM)};
\]

\(F1;\)

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\(i \leftarrow j \times 64\)
\(k \leftarrow j \times 32\)

IF \(k1[j] OR \text{\*no writemask\*}\)

THEN

\[
\text{DEST[i+63:i]} \leftarrow \text{Convert\_Single\_Precision\_To\_UQuadInteger(SRC[k+31:k])}
\]

ELSE

IF \text{\*merging-masking*}; merging-masking

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

ELSE \text{\*zeroing-masking}

\[
\text{DEST[i+63:i]} \leftarrow 0
\]

FI

\(F1;\)

ENDFOR

\[
\text{DEST[MAX\_VL-1:VL]} \leftarrow 0
\]

**VCVTPS2UQQ (EVEX encoded versions) when src operand is a memory source**

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

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\(i \leftarrow j \times 64\)
\(k \leftarrow j \times 32\)

IF \(k1[j] OR \text{\*no writemask\*}\)

THEN

IF \(\text{EVEX.b == 1}\)

THEN

\[
\text{DEST[i+63:i]} \leftarrow \text{Convert\_Single\_Precision\_To\_UQuadInteger(SRC[31:0])}
\]

ELSE

\[
\text{DEST[i+63:i]} \leftarrow \text{Convert\_Single\_Precision\_To\_UQuadInteger(SRC[k+31:k])}
\]

\(F1;\)

ELSE

IF \text{\*merging-masking*}; merging-masking

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

ELSE \text{\*zeroing-masking}

\[
\text{DEST[i+63:i]} \leftarrow 0
\]

FI

\(F1;\)

ENDFOR

\[
\text{DEST[MAX\_VL-1:VL]} \leftarrow 0
\]
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPS2UQQ __m512i _mm512_cvtps_epu64( __m512 a);
VCVTPS2UQQ __m512i _mm512_mask_cvtps_epu64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_maskz_cvtps_epu64( __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_cvt_roundps_epu64( __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_mask_cvt_roundps_epu64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_maskz_cvt_roundps_epu64( __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i _mm256_cvtps_epu64( __m256 a);
VCVTPS2UQQ __m256i _mm256_mask_cvtps_epu64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i _mm256_maskz_cvtps_epu64( __mmask8 k, __m256 a);
VCVTPS2UQQ __m128i _mm128_cvtps_epu64( __m128 a);
VCVTPS2UQQ __m128i _mm128_mask_cvtps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i _mm128_maskz_cvtps_epu64( __mmask8 k, __m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3

#UD If EVEX.vvvv != 1111B.
VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.6F3.0F.W1 E6 /r VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed quadword integers from xmm2/m128/m64bcst to packed double-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.6F3.0F.W1 E6 /r VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.6F3.0F.W1 E6 /r VCVTQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Description**

Converts packed quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

**Operation**

**VCVTQQ2PD (EVEX2 encoded versions) when src operand is a register**

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

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

THEN

SET_RMEVEX,RC;
ELSE

SET_RMMXCSR, RM;
FI;

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:j] ← Convert QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:j])

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:j] ← 0

FI

FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0
VCVTQQ2PD (EVEX encoded versions) when src operand is a memory source

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

FOR \( j \leftarrow 0 \) TO \( KL-1 \)

\( i \leftarrow j \times 64 \)

IF \( k1[j] \) OR "*no writemask*"

THEN

```
IF (EVEX.b == 1)
     THEN
         DEST[i+63:i] ← Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
     ELSE
         DEST[i+63:i] ← Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
     FI;
```

ELSE

```
IF "*merging-masking*" ; merging-masking
     THEN "DEST[i+63:i] remains unchanged*"
     ELSE ; zeroing-masking
         DEST[i+63:i] ← 0
     FI
```

FI

ENDFOR

\( DEST[MAX_VL-1:VL] \leftarrow 0 \)

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

```
VCVTQQ2PD __m512d _mm512_cvtepi64_pd( __m512i a);
VCVTQQ2PD __m512d _mm512_mask_cvtepi64_pd( __m512d s, __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_maskz_cvtepi64_pd( __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_cvt_roundepi64_pd( __m512i a, int r);
VCVTQQ2PD __m512d _mm512_mask_cvtepi64_pd( __m512d s, __mmask8 k, __m512i a, int r);
VCVTQQ2PD __m512d _mm512_maskz_cvtepi64_pd( __mmask8 k, __m512i a, int r);
```

**SIMD Floating-Point Exceptions**

**Precision**

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2

\#UD  

If EVEX.vvvv != 1111B.
VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed quadword integers from xmm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed quadword integers from ymm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W1 5B /r VCVTQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst(ER)</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed quadword integers from zmm2/mem to eight packed single-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed quadword integers in the source operand (second operand) to packed single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a YMM/XMM/XMM (lower 64 bits) register conditionally updated with writemask k1.

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

**Operation**

**VCVTQQ2PS (EVEX encoded versions) when src operand is a register**

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

FOR j ← 0 TO KL-1

i ← j * 64
k ← j * 32

IF k1[j] OR *no writemask*
THEN DEST[k+31:k] ← Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[k+31:k] remains unchanged*
ELSE ; zeroing-masking
DEST[k+31:k] ← 0
FI

FI
ENDFOR

DEST[MAX_VL-1:VL/2] ← 0
VCVTQQ2PS (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
   i ← j * 64
   k ← j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b == 1)
            THEN
               DEST[k+31:k] ← Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
            ELSE
               DEST[k+31:k] ← Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
            FI;
         ELSE
            IF *merging-masking* ; merging-masking
               THEN *DEST[k+31:k] remains unchanged*
            ELSE ; zeroing-masking
               DEST[k+31:k] ← 0
            FI
         FI;
   ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTQQ2PS __m256 _mm512_cvtepi64_ps( __m512i a);
VCVTQQ2PS __m256 _mm512_mask_cvtepi64_ps( __m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_maskz_cvtepi64_ps( __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_cvt_roundepi64_ps( __m512i a, int r);
VCVTQQ2PS __m256 _mm512_mask_cvt_roundepi64_ps( __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m256 _mm512_maskz_cvt_roundepi64_ps( __mmask8 k, __m512i a, int r);

SIMD Floating-Point Exceptions
Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2
#UD IF EVEX.vvvv != 1111B.
VCVTSD2USI—Convert Scalar Double-Precision Floating-Point Value to Unsigned Doubleword Integer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F2.0F:W0 79 /r VCVTD2USI r32, xmm1/m64{er}</td>
<td>T1F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.</td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F:W1 79 /r VCVTD2USI r64, xmm1/m64{er}</td>
<td>T1F</td>
<td>V/N.E.1</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.</td>
</tr>
</tbody>
</table>

NOTES:
1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1F</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Converts a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

Operation
VCVTSD2USI (EVEX encoded version)
IF (SRC *is register*) AND (EVEX.b = 1)
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI
IF 64-Bit Mode and OperandSize = 64
    THEN DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
    ELSE DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
FI

Intel C/C++ Compiler Intrinsic Equivalent
VCVTSD2USI unsigned int _mm_cvtsd_u32(__m128d);
VCVTSD2USI unsigned int _mm_cvtsd_roundsd_u32(__m128d, int r);
VCVTSD2USI unsigned __int64 _mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned __int64 _mm_cvtsd_roundsd_u64(__m128d, int r);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E3NF.
VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F3.0F.w0 79 /r VCVTSS2USI r32, xmm1/m32[er]</td>
<td>T1F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.w1 79 /r VCVTSS2USI r64, xmm1/m32[er]</td>
<td>T1F</td>
<td>V/N.E.¹</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.</td>
</tr>
</tbody>
</table>

NOTES:
1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1F</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Converting a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^{w} - 1$ is returned, where $w$ represents the number of bits in the destination format.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

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

Operation
VCVTSS2USI (EVEX encoded version)
IF (SRC *is register*) AND (EVEX.b = 1)  
THEN  
SET_RM(EVEX.RC);  
ELSE  
SET_RM(MXCSR.RM);  
FI;  
IF 64-bit Mode and OperandSize = 64  
THEN  
DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);  
ELSE  
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);  
FI;

Intel C/C++ Compiler Intrinsic Equivalent
VCVTSS2USI unsigned __m128 _mm_cvtss_u32( __m128 a);  
VCVTSS2USI unsigned __m128 _mm_cvt_roundss_u32( __m128 a, int r);  
VCVTSS2USI unsigned __int64 _mm_cvtss_u64( __m128 a);  
VCVTSS2USI unsigned __int64 _mm_cvt_roundss_u64( __m128 a, int r);
SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E3NF.
VCVTTPD2QQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.w1 7A/r VCVTTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double-precision floating-point values from zmm2/m128/m64bcst to two packed quadword integers in zmm1 using truncation with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.w1 7A/r VCVTTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 using truncation with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.w1 7A/r VCVTTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double-precision floating-point values from zmm2/m512 to eight packed quadword integers in zmm1 using truncation with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value ($2^{w-1}$, where $w$ represents the number of bits in the destination format) is returned.

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

**Operation**

**VCVTTPD2QQ (EVEX encoded version) when src operand is a register**

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

FOR $j \leftarrow 0$ TO KL-1

$i \leftarrow j * 64$

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

THEN DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAX_VL-1:VL] ← 0
VCVTTPD2QQ (EVEX encoded version) when src operand is a memory source

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

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged* 
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR
DEST[MAX_VL-1:VL] ← 0

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

VCVTTPD2QQ __m512i _mm512_cvttpd_epi64( __m512d a);
VCVTTPD2QQ __m512i _mm512_mask_cvttpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_maskz_cvttpd_epi64( __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_cvtt_roundpd_epi64( __m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_mask_cvtt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_maskz_cvtt_roundpd_epi64( __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m256i _mm256_mask_cvttpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i _mm256_maskz_cvttpd_epi64( __mmask8 k, __m256d a);
VCVTTPD2QQ __m128i _mm_mask_cvttpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i _mm_maskz_cvttpd_epi64( __mmask8 k, __m128d a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTTPD2UDQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

<table>
<thead>
<tr>
<th>Opcode Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W1 78 1b EVXTPPDUDQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>FV/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.0F.W1 78 02 1b EVXTPPDUDQ xmm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>FV/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.0F.W1 78 1b EVXTPPDUDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}</td>
<td>FV/V</td>
<td>AVX512F</td>
<td>Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts with truncation packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value 2^w – 1 is returned, where w represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAX_VL-1:256) of the corresponding destination are zeroed.

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

VCVTTPD2UDQ (EVEX encoded versions) when src2 operand is a register

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

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
  THEN
    DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

VCVTTPD2UDQ (EVEX encoded versions) when src operand is a memory source

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

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
    ELSE
      DEST[i+31:i] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTTPD2UDQ __m256i _mm256_cvtt_roundpd_epi32(__m512d a);
VCVTTPD2UDQ __m256i _mm256_mask_cvtt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm256_maskz_cvtt_roundpd_epi32(__mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm256_cvtt_roundpd_epi32(__m512d a, int sae);
VCVTTPD2UDQ __m256i _mm256_mask_cvtt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm256_maskz_cvtt_roundpd_epi32(__mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m128i _mm256_mask_cvtt_roundpd_epi32(__m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm256_maskz_cvtt_roundpd_epi32(__mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm_mask_cvtt_roundpd_epi32(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __m128i _mm_maskz_cvtt_roundpd_epi32(__mmask8 k, __m128d a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvv != 111B.
VCVTTPD2UQQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W1</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed unsigned quadword integers in xmm1 using truncation with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed unsigned quadword integers in ymm1 using truncation with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

**EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

**Operation**

**VCVTTPD2UQQ (EVEX encoded versions) when src operand is a register**

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

FOR $j \leftarrow 0$ TO $KL - 1$

\[ i \leftarrow j \times 64 \]

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

THEN $\text{DEST}[i+63:j] \leftarrow \text{Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger\_Truncate}(\text{SRC}[i+63:j])$

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

\[ \text{DEST}[i+63:j] \leftarrow 0 \]

FI

ENDFOR

\[ \text{DEST}[\text{MAX\_VL-1:VL}] \leftarrow 0 \]
VCVTTPD2UQQ (EVEX encoded versions) when src operand is a memory source

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

FOR j ← 0 TO KL-1
i ← j * 64
IF k1[j] OR *no writemask*
THEN
IF (EVEX.b == 1)
THEN
DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[63:0])
ELSE
DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] ← 0
FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2UQQ _mm<size>[_mask[z]]_cvtt[_round]pd_epu64
VCVTTPD2UQQ __m512i __m512_cvtpd_epu64(__m512d a);
VCVTTPD2UQQ __m512i __m512_mask_cvtpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __m512_maskz_cvtpd_epu64(__mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __m512_cvtt_roundpd_epu64(__m512d a, int sae);
VCVTTPD2UQQ __m512i __m512_mask_cvtt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m128i __m128_cvtpd_epu64(__m128d a);
VCVTTPD2UQQ __m128i __m128_maskz_cvtpd_epu64(__mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i __m128_mask_cvtpd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i __m128_maskz_cvtpd_epu64(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTTPS2UDQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.0F.W0 78 /r VCVTTPS2UDQ xmm1 {k1}[z], xmm2/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W0 78 /r VCVTTPS2UDQ ymm1 {k1}[z], ymm2/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W0 78 /r VCVTTPS2UDQ zmm1 {k1}[z], zmm2/m512/m32bcst{sae}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts with truncation packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value \(2^w - 1\) is returned, where \(w\) represents the number of bits in the destination format.

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

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

Operation

VCVTTPS2UDQ (EVEX encoded versions) when src operand is a register

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

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI
ENDFOR

DEST[MAX_VL-1:VL] ← 0
VCVTTPS2UDQ (EVEX encoded versions) when src operand is a memory source

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

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+31:i] ←
          Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN 'DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
  ENDFOR

DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2UDQ __m512i _mm512_cvttps_epu32( __m512 a);
VCVTTPS2UDQ __m512i _mm512_mask_cvttps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_maskz_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_cvtt_roundps_epu32( __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_mask_cvtt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_maskz_cvtt_roundps_epu32( __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i _mm256_mask_cvttps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i _mm256_maskz_cvttps_epu32( __mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i _mm_mask_cvttps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i _mm_maskz_cvttps_epu32( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 7A /r VCVTTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 7A /r VCVTTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 7A /r VCVTTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**
Converting with truncation packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^{w-1}\), where w represents the number of bits in the destination format, is returned.

**EVEX encoded versions:** The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

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

**Operation**

```plaintext
VCVTTPS2QQ (EVEX encoded versions) when src operand is a register
KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
i ← j * 64
k ← j * 32
IF k1[j] OR *no writemask*
  THEN DEST[i+63:j] ← Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:j] remains unchanged*
  ELSE ; zeroing-masking
    DEST[i+63:j] ← 0
  FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
```

5-66 Vol. 2C VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values
VCVTTPS2QQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ← Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+63:i] ← Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI
  END FOR

DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTPS2QQ __m512i _mm512_cvttps_epi64(__m256 a);
VCVTTPS2QQ __m512i _mm512_mask_cvttps_epi64(__m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_maskz_cvttps_epi64(__mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_cvtt_roundps_epi64(__m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_maskz_cvtt_roundps_epi64(__mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i _mm256_mask_cvttps_epi64(__m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m256i _mm256_maskz_cvttps_epi64(__mmask16 k, __m256 a);
VCVTTPS2QQ __m128i _mm128_mask_cvttps_epi64(__m256i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm128_maskz_cvttps_epi64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E3.
#UD If EVEX.vvvv != 1111B.
VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 78 /r</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 78 /r</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 78 /r</td>
<td>HV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts with truncation up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w – 1$ is returned, where $w$ represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

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

Operation

VCVTTPS2UQQ (EVEX encoded versions) when src operand is a register

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

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:j] ← Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:j] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:j] ← 0
      FI
  FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VCVTTPS2UQQ (EVEX encoded versions) when src operand is a memory source

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

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k[i] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ← Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[31:0])
        ELSE
          DEST[i+63:i] ← Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2UQQ _mm<size>_mask[<z>]._cvtt[_round]ps_epu64
VCVTTPS2UQQ _m512i _mm512_cvttps_epu64(__m256 a);
VCVTTPS2UQQ _m512i _mm512_mask_cvttps_epu64(__m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ _m512i _mm512_maskz_cvttps_epu64(__mmask16 k, __m256 a);
VCVTTPS2UQQ _m512i _mm512_cvtt_roundps_epu64(__m256 a, int sae);
VCVTTPS2UQQ _m512i _mm512_mask_cvtt_roundps_epu64(__m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ _m512i _mm512_maskz_cvtt_roundps_epu64(__mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ _m256i _mm256_cvttps_epu64(__m256i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ _m256i _mm256_mask_cvttps_epu64(__mmask8 k, __m128 a);
VCVTTPS2UQQ _m128i _mm_mask_cvttps_epu64(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ _m128i _mm_maskz_cvttps_epu64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.
VCVTTSD2USI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Unsigned Integer

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F2.0F.W0 7B /r</td>
<td>T1F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.</td>
</tr>
<tr>
<td>VCVTTSD2USI r32, xmm1/m64{sa}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.F2.0F.W1 7B /r</td>
<td>T1F</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.</td>
</tr>
<tr>
<td>VCVTTSD2USI r64, xmm1/m64{sa}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1F</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts with truncation a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value 2^w – 1 is returned, where w represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

### Operation

**VCVTTSD2USI (EVEX encoded version)**

IF 64-Bit Mode and OperandSize = 64
  THEN DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
  ELSE DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);

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

VCVTTSD2USI unsigned int _mm_cvttsd_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvttsd_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvttsd_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvttsd_roundsd_u64(__m128d, int sae);

### SIMD Floating-Point Exceptions

Invalid, Precision

### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.
VCVTTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.F3.0F.W0 78 /r</td>
<td>T1F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.</td>
</tr>
<tr>
<td>VCVTTSS2USI r32, xmm1/m32{sae}</td>
<td>T1F</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>EVEX.LIG.F3.0F.W1 78 /r</td>
<td>T1F</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.</td>
</tr>
<tr>
<td>VCVTTSS2USI r64, xmm1/m32{sae}</td>
<td>T1F</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1F</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts with truncation a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

**EVEX.W1 version:** promotes the instruction to produce 64-bit data in 64-bit mode.

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

VCVTSS2USI (EVEX encoded version)
IF 64-bit Mode andOperandSize = 64
THEN
  DEST[63:0]  Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
ELSE
  DEST[31:0]  Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
FI;

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned int _mm_cvtsu_s_u32(__m128 a);
VCVTSS2USI unsigned int _mm_cvts_round_s_u32(__m128 a, int sae);
VCVTSS2USI unsigned __int64 _mm_cvtsu_s_u64(__m128 a);
VCVTSS2USI unsigned __int64 _mm_cvts_round_s_u64(__m128 a, int sae);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E3NF.
VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F.W0 7A /r</td>
<td>HV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert two packed unsigned doubleword integers from ymm2/m64/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W0 7A /r</td>
<td>HV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W0 7A /r</td>
<td>HV V/V</td>
<td>AVX512F</td>
<td>Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Converts packed unsigned doubleword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).
The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Attempt to encode this instruction with EVEX embedded rounding is ignored.

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

Operation

VCVTUDQ2PD (EVEX encoded versions) when src operand is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
            FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VCVTUDQ2PD (EVCX encoded versions) when src operand is a memory source

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

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+63:i] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
  FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PD __m512d _mm512_cvtepu32_pd( __m256i a);
VCVTUDQ2PD __m512d _mm512_mask_cvtepu32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d _mm512_maskz_cvtepu32_pd( __mmask8 k, __m256i a);
VCVTUDQ2PD __m256d _mm256_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m256d _mm256_mask_cvtepu32_pd( __m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d _mm256_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m128d _mm_mask_cvtepu32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_maskz_cvtepu32_pd( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEK-encoded instructions, see Exceptions Type E5.

#UD If EVEX.vvvv != 1111B.
VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.0F:W0 7A /r</td>
<td>V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single-precision floating-point values in xmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F2.0F:W0 7A /r</td>
<td>V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single-precision floating-point values in ymm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F2.0F:W0 7A /r</td>
<td>V</td>
<td>AVX512F</td>
<td>Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Converts packed unsigned doubleword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

### Operation

**VCVTUDQ2PS (EVEX encoded version) when src operand is a register**

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

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

THEN

    SET_RM(EVEX.RC);

ELSE

    SET_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

    i ← j * 32

    IF k1[j] OR *no writemask*

    THEN DEST[i+31:i] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])

    ELSE

    IF *merging-masking* ; merging-masking

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

    ELSE ; zeroing-masking

    DEST[i+31:i] ← 0

    FI

ENDFOR

DEST[MAX_VL-1:VL] ← 0
VCVTUDQ2PS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0])
        ELSE
          DEST[i+31:i] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
  ENDFOR
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTUDQ2PS __m512 _mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m512 _mm512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_cvt_roundepu32_ps( __m512i a, int r);
VCVTUDQ2PS __m512 _mm512_mask_cvt_roundepu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 _mm512_maskz_cvt_roundepu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 _mm256_cvtepu32_ps( __m256i a);
VCVTUDQ2PS __m256 _mm256_mask_cvtepu32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 _mm256_maskz_cvtepu32_ps( __mmask8 k, __m256i a);
VCVTUDQ2PS __m128 _mm128_cvtepu32_ps( __m128i a);
VCVTUDQ2PS __m128 _mm128_mask_cvtepu32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 _mm128_maskz_cvtepu32_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions
Precision

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E2.
#UD If EVEX:vvvv != 1111B.
VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F.W1 7A /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W1 7A /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W1 7A /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts packed unsigned quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

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

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

Operation

VCVTUQQ2PD (EVEX encoded version) when src operand is a register

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

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

THEN

SET_RM(EVEX.RC);
ELSE

SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] ← Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0
VCVTUQQ2PD (EVEX encoded version) when src operand is a memory source

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ← Convert_UQuaDivision_To_Double_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+63:i] ← Convert_UQuaDivision_To_Double_Precision_Floating_Point(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking*
          THEN *DEST[i+63:i] remains unchanged*
        ELSE
          DEST[i+63:i] ← 0
        FI
      FI
  ENDFOR

DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PD __m512d _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d _mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_mask_cvt_roundepu64_ps( __m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d _mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_cvt_roundepu64_ps( __m256i a, int r);
VCVTUQQ2PD __m256d _mm256_mask_cvt_roundepu64_ps( __m256d s, __mmask8 k, __m256i a, int r);
VCVTUQQ2PD __m256d _mm256_maskz_cvt_roundepu64_ps( __mmask8 k, __m256i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

**opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| EVEX.128.F2.0F:W1 7A /
VCVTUQQ2PS xmm1 {k1}{z},
xmm2/m128/m64bcst | FV | V/V | AVX512VL
AVX512DQ | Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single-precision floating-point values in zmm1 with writemask k1. |
| EVEX.256.F2.0F:W1 7A /
VCVTUQQ2PS xmm1 {k1}{z},
ymm2/m256/m64bcst | FV | V/V | AVX512VL
AVX512DQ | Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single-precision floating-point values in xmm1 with writemask k1. |
| EVEX.512.F2.0F:W1 7A /
VCVTUQQ2PS ymm1 {k1}{z},
zmm2/m512/m64bcst(er) | FV | V/V | AVX512DQ | Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single-precision floating-point values in ymm1 with writemask k1. |

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts packed unsigned quadword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

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

**Operation**

**VCVTUQQ2PS (EVEX encoded version) when src operand is a register**

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

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

THEN

SET_RM(EVEX.RC);

ELSE

SET_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j * 32

k ← j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] ← Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX_VL-1:VL/2] ← 0
VCVTUQQ2PS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ← Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+31:i] ← Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PS __m256 _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PS __m256 _mm512_mask_cvtepu64_ps( __m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PS __m256 _mm512_mask_cvt_roundepu64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 _mm512_cvtepu64_ps( __m256i a);
VCVTUQQ2PS __m256 _mm_mask_cvtepu64_ps( __m256i s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m256 _mm_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PS __m128 _mm_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm_cvt_roundepu64_ps( __m128i a);
VCVTUQQ2PS __m128 _mm_mask_cvt_roundepu64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 _mm_maskz_cvt_roundepu64_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.
VCVTUSI2SD—Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.F2.0F.W0 7B / r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one unsigned doubleword integer from r/m32 to one double-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F2.0F.W1 7B / r</td>
<td>T1S</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Convert one unsigned quadword integer from r/m64 to one double-precision floating-point value in xmm1.</td>
</tr>
</tbody>
</table>

NOTES:
1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX:vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.
EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

Operation

VCVTUSI2SD (EVEX encoded version)

IF (SRC2 *is register*) AND (EVEX.b = 1)
THEN
SET_RM(EVEX.RC);
ELSE
SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
DEST[63:0] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
DEST[63:0] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VCVTUSI2SD __m128d _mm_cvtu32_sd(__m128d s, unsigned a);
VCVTUSI2SD __m128d _mm_cvtu64_sd(__m128d s, unsigned __int64 a);
VCVTUSI2SD __m128d _mm_cvt_roundu64_sd(__m128d s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions
Precision

Other Exceptions
See Exceptions Type E3NF if W1, else type E10NF.
VCVTUSI2SS—Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.F3.0F.W0 7B /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.LIG.F3.0F.W1 7B /r</td>
<td>T1S</td>
<td>V/N.E.1</td>
<td>AVX512F</td>
<td>Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.</td>
</tr>
</tbody>
</table>

**NOTES:**

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

**Description**

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

**EVEX.W1 version:** promotes the instruction to use 64-bit input value in 64-bit mode.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Operation

VCVTUSI2SS (EVEX encoded version)

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

IF 64-Bit Mode And OperandSize = 64
    THEN
        DEST[31:0] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0]);
    FI;

DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SS __m128 _mm_cvtu32_ss( __m128 s, unsigned a);
VCVTUSI2SS __m128 _mm_cvt_roundu32_ss( __m128 s, unsigned a, int r);
VCVTUSI2SS __m128 _mm_cvtu64_ss( __m128 s, unsigned __int64 a);
VCVTUSI2SS __m128 __m_cvt_roundu64_ss( __m128 s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type E3NF.
VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 42 /r ib VDBPSADBW xmm1 {k1}{z}, xmm2, xmm3/m128, imm8</td>
<td>FVM V/V AVX512VL AVX512BW</td>
<td>Compute packed SAD word results of unsigned bytes in dword block from xmm2 with unsigned bytes of dword blocks transformed from xmm3/m128 using the shuffle controls in imm8. Results are written to xmm1 under the writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 42 /r ib VDBPSADBW ymm1 {k1}{z}, ymm2, ymm3/m256, imm8</td>
<td>FVM V/V AVX512VL AVX512BW</td>
<td>Compute packed SAD word results of unsigned bytes in dword block from ymm2 with unsigned bytes of dword blocks transformed from ymm3/m256 using the shuffle controls in imm8. Results are written to ymm1 under the writemask k1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 42 /r ib VDBPSADBW zmm1 {k1}{z}, zmm2, zmm3/m512, imm8</td>
<td>FVM V/V AVX512BW</td>
<td>Compute packed SAD word results of unsigned bytes in dword block from zmm2 with unsigned bytes of dword blocks transformed from zmm3/m512 using the shuffle controls in imm8. Results are written to zmm1 under the writemask k1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each superblock of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see “Tmp1” in Figure 5-8, is constructed from the second source operand and the imm8 byte as shuffle control to select dword elements within a 128-bit lane of the second source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 0 and 1 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 0.

- The next two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from the intermediate vector Tmp1 with a second stationary dword element in the corresponding qword superblock of the first source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 2 and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.

- The intermediate vector is constructed in 128-bits lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the intermediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.
Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBW
**Operation**

**VDBPSADBW (EVEX encoded versions)**

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

Selection of quadruplets:

FOR I = 0 to VL step 128
  TMP1[I+31:I] ← select (SRC2[I+127:I], imm8[1:0])
  TMP1[I+63:I+32] ← select (SRC2[I+127:I], imm8[3:2])
  TMP1[I+95:I+64] ← select (SRC2[I+127:I], imm8[5:4])
  TMP1[I+127:I+96] ← select (SRC2[I+127:I], imm8[7:6])
END FOR

SAD of quadruplets:

FOR I = 0 to VL step 64
  TMP_DEST[I+15:I] ← abs(SRC1[I+7:I] - TMP1[I+7:I]) +
  abs(SRC1[I+15:I+8] - TMP1[I+15:I+8]) +
  abs(SRC1[I+23:I+16] - TMP1[I+23:I+16]) +
  abs(SRC1[I+31:I+24] - TMP1[I+31:I+24])

  TMP_DEST[I+31:I+16] ← abs(SRC1[I+7:I] - TMP1[I+15:I+8]) +
  abs(SRC1[I+15:I+8] - TMP1[I+23:I+16]) +
  abs(SRC1[I+23:I+16] - TMP1[I+31:I+24]) +
  abs(SRC1[I+31:I+24] - TMP1[I+39:I+32])

  abs(SRC1[I+47:I+40] - TMP1[I+31:I+24]) +
  abs(SRC1[I+55:I+48] - TMP1[I+39:I+32]) +
  abs(SRC1[I+63:I+56] - TMP1[I+47:I+40])

  abs(SRC1[I+47:I+40] - TMP1[I+39:I+32]) +
  abs(SRC1[I+55:I+48] - TMP1[I+47:I+40]) +
  abs(SRC1[I+63:I+56] - TMP1[I+55:I+48])
ENDFOR

FOR J ← 0 TO KL-1
  I ← j * 16
  IF k1[J] OR *no writemask*
    THEN DEST[I+15:I] ← TMP_DEST[I+15:I]
  ELSE
    IF *merging-masking*
      THEN *DEST[I+15:I] remains unchanged*
    ELSE
      ELSE *zeroing-masking*
        DEST[I+15:I] ← 0
    FI
  FI
ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VDBPSADBw __m512i _mm512_dbsad_epu8(__m512i a, __m512i b);
VDBPSADBw __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b);
VDBPSADBw __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b);
VDBPSADBw __m256i _mm256_dbsad_epu8(__m256i a, __m256i b);
VDBPSADBw __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b);
VDBPSADBw __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b);
VDBPSADBw __m128i _mm_dbsad_epu8(__m128i a, __m128i b);
VDBPSADBw __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b);
VDBPSADBw __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4NF.nb.
VEXPANDPD—Load Sparse Packed Double-Precision Floating-Point Values from Dense Memory

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 88 /r VEXPANDPD xmm1 [k1][z], xmm2/m128</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 88 /r VEXPANDPD ymm1 [k1][z], ymm2/m256</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 88 /r VEXPANDPD zmm1 [k1][z], zmm2/m512</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description
Expand (load) up to 8/4/2, contiguous, double-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation
VEXPANDPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
k ← 0
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      DEST[i+63:i] ← SRC[k+63:k];
      k ← k + 64
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[i+63:i] ← 0
    FI
  FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VEXPANDPD __m512d _mm512_mask_expand_pd( __m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_maskz_expand_pd( __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_maskz_expand_loadu_pd( __m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d _mm512_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_mask Expand_pd( __m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_maskz_expand_pd( __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_maskz_expand_loadu_pd( __m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m128d _mm_mask_expand_pd( __m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_maskz_expand_pd( __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_maskz_expand_loadu_pd( __m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d _mm_maskz_expandloadu_pd( __mmask8 k, void * a);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E4.nb.

#UD If EVEX.vvvv != 111B.
VEXPANDPS—Load Sparse Packed Single-Precision Floating-Point Values from Dense Memory

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 88 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 88 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 88 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Expand (load) up to 16/8/4, contiguous, single-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register; the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

VEXPANDPS (EVEX encoded versions)

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

\( k \leftarrow 0 \)

FOR \( j \leftarrow 0 \) TO KL-1

\( i \leftarrow j \times 32 \)

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

THEN

\( \text{DEST}[i+31:i] \leftarrow \text{SRC}[k+31:k]; \)

\( k \leftarrow k + 32 \)

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

\( \text{DEST}[i+31:i] \leftarrow 0 \)

Fi

ENDFOR

\( \text{DEST}[\text{MAX_VL}-1:VL] \leftarrow 0 \)
Intel C/C++ Compiler Intrinsic Equivalent

VEXPANDPS __m512 __m512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 __m512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 __m512_mask_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 __m512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 __m256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 __m256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 __m256_mask_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 __m256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 __m128_mask_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 __m128_maskz_expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 __m128_mask_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 __m128_maskz_expandloadu_ps( __mmask8 k, void * a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.nb.

#UD

If EVEX.vvvv != 1111B.
VERR/VERW—Verify a Segment for Reading or Writing

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 00 /4</td>
<td>VERR r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set ZF=1 if segment specified with r/m16 can be read.</td>
</tr>
<tr>
<td>0F 00 /5</td>
<td>VERW r/m16</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Set ZF=1 if segment specified with r/m16 can be written.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not NULL.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment’s DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector’s RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

Operation

IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))
    THEN ZF ← 0; Fl;

Read segment descriptor;

IF SegmentDescriptor(DescriptorType) = 0 (* System segment *)
or (SegmentDescriptor(Type) ≠ conforming code segment)
and (CPL > DPL) or (RPL > DPL)
    THEN
        ZF ← 0;
    ELSE
        IF ((Instruction = VERR) and (Segment readable))
        or ((Instruction = VERW) and (Segment writable))
            THEN
                ZF ← 1;
        Fl;
        Fl;
**Flags Affected**
The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

**Protected Mode Exceptions**
The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

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

**Real-Address Mode Exceptions**

- **#UD** The VERR and VERW instructions are not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

- **#UD** The VERR and VERW instructions are not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.
VEXP2PD—Approximation to the Exponential $2^x$ of Packed Double-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W1 C8 / r VEXP2PD zmm1 [k1][z], zmm2/m512/m64bcst (sae)</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the approximate base-2 exponential evaluation of the double-precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than $2^{-23}$ of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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


**Operation**

VEXP2PD

(\(KL, VL\) = (8, 512))

FOR \(j \leftarrow 0\) TO \(KL-1\)

\(i \leftarrow j \times 64\)

IF k1[j] OR *no writemask* THEN

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

THEN DEST[i+63:j] \(\leftarrow\) EXP2_23_DP(SRC[63:0])

ELSE DEST[i+63:j] \(\leftarrow\) EXP2_23_DP(SRC[i+63:j])

FI;

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:j] \(\leftarrow\) 0

FI;

FI;

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

VEXP2PD __m512d __mm512_exp2a23_round_pd (__m512d a, int sae);
VEXP2PD __m512d __mm512_mask_exp2a23_round_pd (__m512d a, __mmask8 m, __m512d b, int sae);
VEXP2PD __m512d __mm512_maskz_exp2a23_round_pd (__mmask8 m, __m512d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Overflow

**Other Exceptions**

See Exceptions Type E2.

**Table 5-4. Special Values Behavior**

<table>
<thead>
<tr>
<th>Source Input</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNaN(src)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>+/-∞</td>
<td>+/-∞</td>
<td></td>
</tr>
<tr>
<td>+/-0</td>
<td>1.0f</td>
<td>Exact result</td>
</tr>
<tr>
<td>-0</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>Integral value N</td>
<td>2^N</td>
<td>Exact result</td>
</tr>
</tbody>
</table>
VEXP2PS—Approximation to the Exponential $2^x$ of Packed Single-Precision Floating-Point Values with Less Than $2^{-23}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 C8 /r</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the exponential $2^x$ (with less than $2^{-23}$ of maximum relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the approximate base-2 exponential evaluation of the single-precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than $2^{-23}$ of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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


**Operation**

**VEXP2PS**

$(KL, VL) = (16, 512)$

FOR $j \leftarrow 0$ TO $KL-1$

$i \leftarrow j \times 32$

IF k1[j] OR *no writemask* THEN

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

THEN DEST[i+31:i] $\leftarrow$ EXP2_23_SP(SRC[31:0])

ELSE DEST[i+31:i] $\leftarrow$ EXP2_23_SP(SRC[i+31:i])

FI;

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] $\leftarrow$ 0

FI;

FI;

ENDFOR;
Intel C/C++ Compiler Intrinsic Equivalent
VEXP2PS __m512 __m512_exp2a23_round_ps (__m512 a, int sae);
VEXP2PS __m512 __m512_mask_exp2a23_round_ps (__m512 a, __mmask16 m, __m512 b, int sae);
VEXP2PS __m512 __m512_maskz_exp2a23_round_ps (__mmask16 m, __m512 b, int sae);

SIMD Floating-Point Exceptions
Invalid (if SNaN input), Overflow

Other Exceptions
See Exceptions Type E2.

Table 5-5. Special Values Behavior

<table>
<thead>
<tr>
<th>Source Input</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNaN(src)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>+/-0</td>
<td>1.0f</td>
<td>Exact result</td>
</tr>
<tr>
<td>-∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>Integral value N</td>
<td>2^ (N)</td>
<td>Exact result</td>
</tr>
</tbody>
</table>
**VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 19 /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 19 /r ib</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Extract 128 bits of packed single-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 19 /r ib</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 128 bits of packed single-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 19 /r ib</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Extract 128 bits of packed double-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 19 /r ib</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 128 bits of packed double-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1B /r ib</td>
<td>T8</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 256 bits of packed single-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1B /r ib</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 256 bits of packed double-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>T2, T4, T8</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

VEXTRACTF128/VEXTRACTF32x4 and VEXTRACTF64x2 extract 128-bits of single-precision floating-point values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTF32x8 and VEXTRACTF64x4 extract 256-bits of double-precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at a 256-bit granular offset specified byimm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 6 bits of the immediate are ignored.

If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Operation

**VEXTRACTF32x4 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256
   CASE (imm8[0]) OF
      0: TMP_DEST[127:0] ← SRC1[127:0]
      1: TMP_DEST[127:0] ← SRC1[255:128]
   ESAC.

FI;

IF VL = 512
   CASE (imm8[1:0]) OF
      00: TMP_DEST[127:0] ← SRC1[127:0]
      01: TMP_DEST[127:0] ← SRC1[255:128]
      10: TMP_DEST[127:0] ← SRC1[383:256]
      11: TMP_DEST[127:0] ← SRC1[511:384]
   ESAC.

FI;

FOR j ← 0 TO 3
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+31:i] ← 0
   FI
ENDFOR

DEST[MAX_VL-1:128] ← 0
VEXTRACTF32x4 (EVEX encoded versions) when destination is memory

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP_DEST[127:0] \leftarrow SRC1[127:0]
1: TMP_DEST[127:0] \leftarrow SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP_DEST[127:0] \leftarrow SRC1[127:0]
01: TMP_DEST[127:0] \leftarrow SRC1[255:128]
10: TMP_DEST[127:0] \leftarrow SRC1[383:256]
11: TMP_DEST[127:0] \leftarrow SRC1[511:384]

ESAC.

FI;

FOR j \leftarrow 0 TO 3

i \leftarrow j * 32

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] \leftarrow TMP_DEST[i+31:i]

ELSE *DEST[i+31:i] remains unchanged* ; merging-masking

FI;

ENDFOR
VEXTRACTF64x2 (EVEX encoded versions) when destination is a register

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP_DEST[127:0] ← SRC1[127:0]
1: TMP_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP_DEST[127:0] ← SRC1[127:0]
01: TMP_DEST[127:0] ← SRC1[255:128]
10: TMP_DEST[127:0] ← SRC1[383:256]
11: TMP_DEST[127:0] ← SRC1[511:384]

ESAC.

FI;

FOR j ← 0 TO 1

i ← j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] ← TMP_DEST[i+63:i]

ELSE

IF *merging-masking* ; merging-masking

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

ELSE *zeroing-masking* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI

ENDIF

DEST[MAX_VL-1:128] ← 0
VEXTRACTF64x2 (EVEX encoded versions) when destination is memory
VL = 256, 512
IF VL = 256
  CASE (imm8[0]) OF
    0: TMP_DEST[127:0] ← SRC1[127:0]
    1: TMP_DEST[127:0] ← SRC1[255:128]
  ESAC.
  FI;
IF VL = 512
  CASE (imm8[1:0]) OF
    00: TMP_DEST[127:0] ← SRC1[127:0]
    01: TMP_DEST[127:0] ← SRC1[255:128]
    10: TMP_DEST[127:0] ← SRC1[383:256]
    11: TMP_DEST[127:0] ← SRC1[511:384]
  ESAC.
  FI;
FOR j ← 0 TO 1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is a register
VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.
FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+31:i] ← 0
      FI
    FI
ENDFOR
DEST[MAX_VL-1:256] ← 0
VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is memory
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VEXTRACTF64x4 (EVEX.512 encoded version) when destination is a register
VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR

DEST[MAX_VL-1:256] ← 0

VEXTRACTF64x4 (EVEX.512 encoded version) when destination is memory
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE ; merging-masking
      *DEST[i+63:i] remains unchanged*
    FI;
ENDFOR
VEXTRACTF128 (memory destination form)
CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.

VEXTRACTF128 (register destination form)
CASE (imm8[0]) OF
  0: DEST[127:0] ← SRC1[127:0]
  1: DEST[127:0] ← SRC1[255:128]
ESAC.
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VEXTRACTF32x4 __m128 _mm512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_mask_extractf32x4_ps(__m512 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_extractf32x4_ps(__m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_mask_extractf32x4_ps(__m256 s, __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_maskz_extractf32x4_ps(__mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_extractf32x8_ps(__m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_mask_extractf32x8_ps(__m512 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_maskz_extractf32x8_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm256_extractf32x8_ps(__m256 a, const int nidx);
VEXTRACTF32x8 __m256 _mm256_mask_extractf32x8_ps(__m256 s, __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x8 __m256 _mm256_maskz_extractf32x8_ps(__mmask8 k, __m256 a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_extractf64x2_pd(__m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_mask_extractf64x2_pd(__m512d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_maskz_extractf64x2_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_extractf64x2_pd(__m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_mask_extractf64x2_pd(__m256d s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_maskz_extractf64x2_pd(__mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_extractf64x4_pd(__m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_mask_extractf64x4_pd(__m512d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_maskz_extractf64x4_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm256_extractf64x4_pd(__m256d a, const int nidx);
VEXTRACTF64x4 __m256d _mm256_mask_extractf64x4_pd(__m256d s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x4 __m256d _mm256_maskz_extractf64x4_pd(__mmask8 k, __m256d a, const int nidx);
VEXTRACTF128 __m128 __m128d_extractf128_ps(__m128 a, int offset);
VEXTRACTF128 __m128d __m256_extractf128_pd(__m256 a, int offset);
VEXTRACTF128 __m128d __m256_extractf128_si256(__m256 a, int offset);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, see Exceptions Type E6NF.
#UD IF VEX.L = 0.
#UD IF VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
### VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 39 /r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 39 /r ib</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 39 /r ib</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 39 /r ib</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 39 /r ib</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3B /r ib</td>
<td>T8</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 3B /r ib</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>T2, T4, T8</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEXTRACTI32x8: The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.
VEXTRACTI64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.
The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.
If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

**Operation**

**VEXTRACTI32x4 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP_DEST[127:0] ← SRC1[127:0]
1: TMP_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP_DEST[127:0] ← SRC1[127:0]
01: TMP_DEST[127:0] ← SRC1[255:128]
10: TMP_DEST[127:0] ← SRC1[383:256]
11: TMP_DEST[127:0] ← SRC1[511:384]

ESAC.

FI;

FOR j ← 0 TO 3

i ← j * 32

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] ← TMP_DEST[i+31:i]

ELSE

IF *merging-masking* ; merging-masking

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

ELSE *zeroing-masking* ; zeroing-masking

DEST[i+31:i] ← 0

FI 

FI

ENDFOR

DEST[MAX_VL-1:128] ← 0

**VEXTRACTI32x4 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP_DEST[127:0] ← SRC1[127:0]
1: TMP_DEST[127:0] ← SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP_DEST[127:0] ← SRC1[127:0]
01: TMP_DEST[127:0] ← SRC1[255:128]
10: TMP_DEST[127:0] ← SRC1[383:256]
11: TMP_DEST[127:0] ← SRC1[511:384]

ESAC.
VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

INSTRUCTION SET REFERENCE, V-Z

FOR j ← 0 TO 3
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VEXTRACTI64x2 (EVEX encoded versions) when destination is a register

VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC1[127:0]
        1: TMP_DEST[127:0] ← SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] ← SRC1[127:0]
        01: TMP_DEST[127:0] ← SRC1[255:128]
        10: TMP_DEST[127:0] ← SRC1[383:256]
        11: TMP_DEST[127:0] ← SRC1[511:384]
    ESAC.
FI;
FOR j ← 0 TO 1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI;
ENDFOR

DEST[MAX_VL-1:128] ← 0
VEXTRACTI64x2 (EVEX encoded versions) when destination is memory
VL = 256, 512
IF VL = 256
  CASE (imm8[0]) OF
    0: TMP_DEST[127:0] ← SRC1[127:0]
    1: TMP_DEST[127:0] ← SRC1[255:128]
  ESAC.
FI;
IF VL = 512
  CASE (imm8[1:0]) OF
    00: TMP_DEST[127:0] ← SRC1[127:0]
    01: TMP_DEST[127:0] ← SRC1[255:128]
    10: TMP_DEST[127:0] ← SRC1[383:256]
    11: TMP_DEST[127:0] ← SRC1[511:384]
  ESAC.
FI;
FOR j ← 0 TO 1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is a register
VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.
FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:256] ← 0
VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is a register

VL = 512

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR

DEST[MAX_VL-1:256] ← 0

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory

CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
**VEXTRACTI128 (memory destination form)**

CASE (imm8[0]) OF
   0: DEST[127:0] ← SRC1[127:0]
   1: DEST[127:0] ← SRC1[255:128]
ESAC.

**VEXTRACTI128 (register destination form)**

CASE (imm8[0]) OF
   0: DEST[127:0] ← SRC1[127:0]
   1: DEST[127:0] ← SRC1[255:128]
ESAC.

**DEST[MAX_VL-1:128] ← 0**

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

VEXTRACTI32x4 __m128i _mm512_extracti32x4_epi32(__m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_extracti32x4_epi32(__m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI128 __m128i _mm256_extracti128_si256(__m256i a, int offset);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.

#UD IF VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VFIXUPIMMPD—Fix Up Special Packed Float64 Values

**Operand Encoding**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W1 54 /r ib VFIXUPIMMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float64 vector xmm1, float64 vector xmm2 and int64 vector xmm3/m128/m64bcst and store the result in xmm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 54 /r ib VFIXUPIMMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float64 vector ymm1, float64 vector ymm2 and int64 vector ymm3/m256/m64bcst and store the result in ymm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 54 /r ib VFIXUPIMMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Perform fix-up of quad-word elements encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

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

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.
Operation

enum TOKEN_TYPE
{
    QNAN_TOKEN <- 0,
    SNAN_TOKEN <- 1,
    ZERO_VALUE_TOKEN <- 2,
    POS_ONE_VALUE_TOKEN <- 3,
    NEG_INF_TOKEN <- 4,
    POS_INF_TOKEN <- 5,
    NEG_VALUE_TOKEN <- 6,
    POS_VALUE_TOKEN <- 7
}

FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8 [7:0])
{
    tsrc[63:0] <- ((src1[62:52] = 0) AND MXCSR.DAZ = 1) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j <- 0;
        SNAN_TOKEN: j <- 1;
        ZERO_VALUE_TOKEN: j <- 2;
        POS_ONE_VALUE_TOKEN: j <- 3;
        NEG_INF_TOKEN: j <- 4;
        POS_INF_TOKEN: j <- 5;
        NEG_VALUE_TOKEN: j <- 6;
        POS_VALUE_TOKEN: j <- 7;
    } ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[63:0] <- dest[63:0]; ; preserve content of DEST
        0001: dest[63:0] <- tsrc[63:0]; ; pass through src1 normal input value, denormal as zero
        0010: dest[63:0] <- QNaN(tsrc[63:0]);
        0011: dest[63:0] <- QNaN_Indefinite;
        0100: dest[63:0] <- -INF;
        0101: dest[63:0] <- +INF;
        0111: dest[63:0] <- -0;
        1000: dest[63:0] <- +0;
        1001: dest[63:0] <- -1;
        1010: dest[63:0] <- +1;
        1011: dest[63:0] <- ½;
        1100: dest[63:0] <- 90.0;
        1101: dest[63:0] <- PI/2;
        1110: dest[63:0] <- MAX_FLOAT;
        1111: dest[63:0] <- MAX_FLOAT;
    } ; end of token_response CASE
; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[63:0];
}
; end of FIXUPIMM_DP()

VFIXUPIMMPD
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
i ← j * 64
IF k1[j] OR *no writemask*
THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
            DEST[i+63:i] ← FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[63:0], imm8 [7:0])
        ELSE
            DEST[i+63:i] ← FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[i+63:i], imm8 [7:0])
    FI;
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE DEST[i+63:i] ← 0 ; zeroing-masking
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
Immediate Control Description:

![Diagram of Immediate Control Description]

**Figure 5-9. VFIXUPIIMMPD Immediate Control Description**

**Intel C/C++ Compiler Intrinsic Equivalent**
- `VFIXUPIIMMPD __m512d _mm512_fixupimm_pd(__m512d a, __m512i tbl, int imm);`
- `VFIXUPIIMMPD __m512d _mm512_mask_fixupimm_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm);`
- `VFIXUPIIMMPD __m512d _mm512_maskz_fixupimm_pd(__mmask8 k, __m512d a, __m512i tbl, int imm);`
- `VFIXUPIIMMPD __m512d _mm512_fixupimm_round_pd(__m512d a, __m512i tbl, int sae);`
- `VFIXUPIIMMPD __m512d _mm512_mask_fixupimm_round_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);`
- `VFIXUPIIMMPD __m512d _mm512_maskz_fixupimm_round_pd(__mmask8 k, __m512d a, __m512i tbl, int imm, int sae);`
- `VFIXUPIIMMPD __m256d _mm256_fixupimm_pd(__m256d a, __m512i tbl, int imm);`
- `VFIXUPIIMMPD __m256d _mm256_mask_fixupimm_pd(__m256d s, __mmask8 k, __m256d a, __m256i tbl, int imm);`
- `VFIXUPIIMMPD __m256d _mm256_maskz_fixupimm_pd(__mmask8 k, __m256d a, __m256i tbl, int imm);`
- `VFIXUPIIMMPD __m128d _mm128_fixupimm_pd(__m128d a, __m128i tbl, int imm);`
- `VFIXUPIIMMPD __m128d _mm128_mask_fixupimm_pd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);`
- `VFIXUPIIMMPD __m128d _mm128_maskz_fixupimm_pd(__mmask8 k, __m128d a, __m128i tbl, int imm);`

**SIMD Floating-Point Exceptions**
- Zero, Invalid

**Other Exceptions**
- See Exceptions Type E2.
VFIXUPIMMPS—Fix Up Special Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 54 /r VFIXUPIMMPS xmm1 [k1] [z], xmm2, xmm3/m128/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 54 /r VFIXUPIMMPS ymm1 [k1] [z], ymm2, ymm3/m256/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 54 /r ib VFIXUPIMMPS zmm1 [k1] [z], zmm2, zmm3/m512/m32bcst{sae}, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Perform fix-up of doubleword elements encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

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

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPS can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below). MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set). MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.
Operation

enum TOKEN_TYPE {
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}

FIXUPIMM_SP ( dest[31:0], src1[31:0], tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[31:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j ← 7;
    } ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] ← dest[31:0]; ; preserve content of DEST
        0001: dest[31:0] ← tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] ← QNaN(tsrc[31:0]);
        0011: dest[31:0] ← QNaN_Indefinite;
        0100: dest[31:0] ← -INF;
        0101: dest[31:0] ← +INF;
        0110: dest[31:0] ← tsrc.sign? -INF : +INF;
        0111: dest[31:0] ← 0;
        1000: dest[31:0] ← +0;
        1001: dest[31:0] ← -1;
        1010: dest[31:0] ← -½;
        1100: dest[31:0] ← 90.0;
        1101: dest[31:0] ← PI/2;
        1110: dest[31:0] ← MAX_FLOAT;
        1111: dest[31:0] ← -MAX_FLOAT;
    } ; end of token_response CASE
The required fault reporting from imm8 is extracted.

; Tokens are mutually exclusive and Tokens priority defines the order.

; Multiple faults related to a single token can occur simultaneously.

IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;

; end fault reporting

return dest[31:0];

} ; end of FIXUPIMM_SP()

VFIXUPIMMPS (EVEX)

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

FOR j ← 0 TO KL-1
i ← j * 32
IF k1[j] OR *no writemask*
THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN
DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[31:0], imm8[7:0])
ELSE
DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[i+31:i], imm8[7:0])
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE DEST[i+31:i] ← 0 ; zeroing-masking
FI
ENDIF;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
Immediate Control Description:

![Figure 5-10. VFIXUPIMMPS Immediate Control Description](image_url)

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

VFIXUPIMMPS __m512 _mm512_fixupimm_ps(__m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_ps(__mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 _mm512_fixupimm_round_ps(__m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_round_ps(__mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m256 _mm256_fixupimm_ps(__m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 _mm256_mask_fixupimm_ps(__m256 s, __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 _mm256_maskz_fixupimm_ps(__mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m128 _mm512_fixupimm_ps(__m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m128 _mm_mask_fixupimm_ps(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 _mm_maskz_fixupimm_ps(__mmask8 k, __m128 a, __m128i tbl, int imm);

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E2.
**VFIXUPIMMSD—Fix Up Special Scalar Float64 Value**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W1 55 /r ib VFIXUPIMMSD xmm1 [k1][r], xmm2, xmm3/m64{sae}, imm8</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Perform a fix-up of the low quadword element encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fix up nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below). MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set). MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

### Operation

```c
enum TOKEN_TYPE
{
    QNAN_TOKEN     \= 0,
    SNAN_TOKEN     \= 1,
    ZERO_VALUE_TOKEN \= 2,
    POS_ONE_VALUE_TOKEN \= 3,
    NEG_INF_TOKEN \= 4,
    POS_INF_TOKEN \= 5,
    NEG_VALUE_TOKEN \= 6,
    POS_VALUE_TOKEN \= 7
}
```
FIXUPIMM_DP (dest[63:0], src1[63:0], tbl3[63:0], imm8[7:0])

\[
tsrc[63:0] \leftarrow ((src1[62:52] = 0) \text{ AND } (MXCSR.DAZ = 1)) ? 0.0 : src1[63:0]
\]

CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j \leftarrow 0;
    SNAN_TOKEN: j \leftarrow 1;
    ZERO_VALUE_TOKEN: j \leftarrow 2;
    POS_ONE_VALUE_TOKEN: j \leftarrow 3;
    NEG_INF_TOKEN: j \leftarrow 4;
    POS_INF_TOKEN: j \leftarrow 5;
    NEG_VALUE_TOKEN: j \leftarrow 6;
    POS_VALUE_TOKEN: j \leftarrow 7;
}

; The required response from src3 table is extracted

\[
\text{token_response}[3:0] = tbl3[3+4*j:4*j];
\]

CASE(token_response[3:0]) {
    0000: dest[63:0] \leftarrow dest[63:0] ; preserve content of DEST
    0001: dest[63:0] \leftarrow tsrc[63:0] ; pass through src1 normal input value, denormal as zero
    0010: dest[63:0] \leftarrow \text{QNaN}(tsrc[63:0]);
    0011: dest[63:0] \leftarrow \text{QNAN_Indefinite};
    0100: dest[63:0] \leftarrow -\text{INF};
    0101: dest[63:0] \leftarrow +\text{INF};
    0110: dest[63:0] \leftarrow \text{tsrc.sign? –INF : +INF};
    0111: dest[63:0] \leftarrow -0;
    1000: dest[63:0] \leftarrow +0;
    1001: dest[63:0] \leftarrow -1;
    1010: dest[63:0] \leftarrow +1;
    1011: dest[63:0] \leftarrow \frac{1}{2};
    1100: dest[63:0] \leftarrow 90.0;
    1101: dest[63:0] \leftarrow \pi/2;
    1110: dest[63:0] \leftarrow \text{MAX_FLOAT};
    1111: dest[63:0] \leftarrow -\text{MAX_FLOAT};
}

; The required fault reporting from imm8 is extracted

; TOKENs are mutually exclusive and TOKENs priority defines the order.

IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;

; end fault reporting

return dest[63:0];
}

; end of FIXUPIMM_DP()
VFIXUPIMMSD (EVEX encoded version)
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])
ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[63:0] remains unchanged* ; zeroing-masking
    ELSE  DEST[63:0] ← 0
    FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

Immediate Control Description:

![Figure 5-11. VFIXUPIMMSD Immediate Control Description](image)

Intel C/C++ Compiler Intrinsic Equivalent
VFIXUPIMMSD __m128d _mm_fixupimm_sd(__m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_sd(__mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_fixupimm_round_sd(__m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_round_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128i tbl, int imm, int sae);

SIMD Floating-Point Exceptions
Zero, Invalid

Other Exceptions
See Exceptions Type E3.
**VFIXUPIMMSS—Fix Up Special Scalar Float32 Value**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 /r ib</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Perform a fix-up of the low doubleword element encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below). MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set). MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

**Operation**

```c
enum TOKEN_TYPE
{
    QNAN_TOKEN    ← 0,
    SNAN_TOKEN    ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```
FIXUPIMM_SP (dest[31:0], src1[31:0], tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] \leftarrow ((src1[30:23] = 0) \text{ AND } (MXCSR.DAZ = 1)) \text{ ? } 0.0 : src1[31:0]
}

CASE(tsrc[63:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j \leftarrow 0;
    SNAN_TOKEN: j \leftarrow 1;
    ZERO_VALUE_TOKEN: j \leftarrow 2;
    POS_ONE_VALUE_TOKEN: j \leftarrow 3;
    NEG_INF_TOKEN: j \leftarrow 4;
    POS_INF_TOKEN: j \leftarrow 5;
    NEG_VALUE_TOKEN: j \leftarrow 6;
    POS_VALUE_TOKEN: j = 7;
} ; end source special CASE(tsrc...)

; The required response from src3 table is extracted
token_response[3:0] = tbl3[3+4*j:4*j];

CASE(token_response[3:0]) {
    0000: dest[31:0] \leftarrow dest[31:0]; ; preserve content of DEST
    0001: dest[31:0] \leftarrow tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
    0010: dest[31:0] \leftarrow QNaN(tsrc[31:0]);
    0011: dest[31:0] \leftarrow QNaN_Indefinite;
    0100: dest[31:0] \leftarrow -INF;
    0101: dest[31:0] \leftarrow +INF;
    0110: dest[31:0] \leftarrow tsrc.sign? –INF : +INF;
    0111: dest[31:0] \leftarrow -0;
    1000: dest[31:0] \leftarrow +0;
    1001: dest[31:0] \leftarrow -1;
    1010: dest[31:0] \leftarrow +1;
    1011: dest[31:0] \leftarrow \frac{1}{2};
    1100: dest[31:0] \leftarrow 90.0;
    1101: dest[31:0] \leftarrow Pi/2;
    1110: dest[31:0] \leftarrow MAX_FLOAT;
    1111: dest[31:0] \leftarrow -MAX_FLOAT;
} ; end of token_response CASE

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[31:0];
} ; end of FIXUPIMM_SP()
**VFIXUIMMSS (EVEX encoded version)**

IF k1[0] OR *no writemask*

THEN DEST[31:0] ← FIXUIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged* ; zeroing-masking

ELSE DEST[31:0] ← 0

FI


DEST[MAX_VL-1:128] ← 0

**Immediate Control Description:**

![Figure 5-12. VFIXUIMMSS Immediate Control Description](image)

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

VFIXUIMMSS _m128 _mm_fixupimm_ss(_m128 a, _m128i tbl, int imm);

VFIXUIMMSS _m128 _mm_mask_fixupimm_ss(_m128 s, _mmask8 k, _m128 a, _m128i tbl, int imm);

VFIXUIMMSS _m128 _mm_maskz_fixupimm_ss(_mmask8 k, _m128 a, _m128i tbl, int imm);

VFIXUIMMSS _m128 _mm_fixupimm_round_ss(_m128 a, _m128i tbl, int imm, int sae);

VFIXUIMMSS _m128 _mm_mask_fixupimm_round_ss(_m128 s, _mmask8 k, _m128 a, _m128i tbl, int imm, int sae);

VFIXUIMMSS _m128 _mm_maskz_fixupimm_round_ss(_mmask8 k, _m128 a, _m128i tbl, int imm, int sae);

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Exceptions Type E3.
### VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 98 /r VFMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 A8 /r VFMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM.reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM.r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM.reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM.r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Perform a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMADD132PD**: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMADD213PD**: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMADD231PD**: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions**: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version**: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version**: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
THEN
   SET_RM(EVEX.RC);
ELSE
   SET_RM(MXCSR.RM);
FI;

FOR j  0 TO KL-1
   i  j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i]  RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+63:i]  0
         FI
      FI;
ENDFOR

DEST[MAX_VL-1:VL]  0

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j  0 TO KL-1
   i  j * 64
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1)
            THEN
               Dest[i+63:i]  RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
               Dest[i+63:i]  RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
         ELSE
            IF *merging-masking* ; merging-masking
               THEN *DEST[i+63:i] remains unchanged*
               ELSE ; zeroing-masking
                  DEST[i+63:i]  0
            FI
         FI;
ENDFOR

DEST[MAX_VL-1:VL]  0
VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                ELSE
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
            FI
        END FOR
    IF (VL = 512) AND (EVEX.b = 1)
        THEN
            SET_RM(EVEX.RC);
        ELSE
            SET_RM(MXCSR.RM);
        FI;
    FOR j ← 0 TO KL-1
        i ← j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+63:i] ←
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            ELSE
                IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
            FI
        END FOR
    DEST[MAX_VL-1:VL] ← 0

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
    FOR j ← 0 TO KL-1
        i ← j * 64
        IF k1[j] OR *no writemask*
            THEN DEST[i+63:i] ←
                RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            ELSE
                IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
            FI
        END FOR
    DEST[MAX_VL-1:VL] ← 0
VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(SRC2[i+63:j]*SRC3[i+63:j] + DEST[i+63:i])
        ELSE
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(SRC2[i+63:j]*SRC3[i+63:j] + DEST[i+63:j])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:j] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:j] ← 0
      FI
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD __m256d _mm256_fmadd_pd(__m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_round_pd(__mmask8 k, __m256d a, __m256d b, __m256d c, int r);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_round_pd(__mmask8 k, __m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_mask_fmadd_pd(__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_maskz_fmadd_round_pd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxPD __m128d _mm_mask3_fmadd_round_pd(__mmask8 k, __m128d a, __m128d b, __m128d c, __mmask8 k);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
### VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 98 /r VFMADD132PS xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 98 /r VFMADD132PS ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 B8 /r VFMADD231PS ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 98 /r VFMADD132PS ymm1, ymm2, ymm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m128/m32bcst, add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 A8 /r VFMADD213PS ymm1, ymm2, ymm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/m128/m32bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 B8 /r VFMADD231PS ymm1, ymm2, ymm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m128/m32bcst, add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 98 /r VFMADD132PS zmm1, zmm2, zmm3/m256/m32bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m256/m32bcst, add to zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 A8 /r VFMADD213PS zmm1, zmm2, zmm3/m256/m32bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, add to zmm3/m256/m32bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 B8 /r VFMADD231PS zmm1, zmm2, zmm3/m256/m32bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m256/m32bcst, add to zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMADD132PS**: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD213PS**: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD231PS**: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions**: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version**: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version**: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
Operation
In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PS** DEST, SRC2, SRC3

If (VEX.128) THEN
   MAXNUM ←4
ElseIf (VEX.256)
   MAXNUM ←8
FI
For i = 0 to MAXNUM-1 {
   n ←32*i;
   DEST[n+31:n] ←RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
If (VEX.128) THEN
   DEST[MAX_VL-1:128] ←0
ElseIf (VEX.256)
   DEST[MAX_VL-1:256] ←0
FI

**VFMADD213PS** DEST, SRC2, SRC3

If (VEX.128) THEN
   MAXNUM ←4
ElseIf (VEX.256)
   MAXNUM ←8
FI
For i = 0 to MAXNUM-1 {
   n ←32*i;
   DEST[n+31:n] ←RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
If (VEX.128) THEN
   DEST[MAX_VL-1:128] ←0
ElseIf (VEX.256)
   DEST[MAX_VL-1:256] ←0
FI

**VFMADD231PS** DEST, SRC2, SRC3

If (VEX.128) THEN
   MAXNUM ←4
ElseIf (VEX.256)
   MAXNUM ←8
FI
For i = 0 to MAXNUM-1 {
   n ←32*i;
   DEST[n+31:n] ←RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
If (VEX.128) THEN
   DEST[MAX_VL-1:128] ←0
ElseIf (VEX.256)
   DEST[MAX_VL-1:256] ←0
FI
VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
        RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
    ELSE
        IF *merging-masking* ; merging-mask
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-mask
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] ←
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
            ELSE
                DEST[i+31:i] ←
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-mask
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-mask
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
        ELSE
          DEST[i+31:i] ←
          RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
    FI;
  ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR \(j \leftarrow 0\) TO \(KL-1\)
  \(i \leftarrow j * 32\)
  IF \(k1[j]\) OR *no writemask*
    THEN
      IF \((EVEX.b = 1)\)
        THEN
          \(DEST[i+31:i] \leftarrow \) RoundFPControl_MXCSR\((SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])\)
        ELSE
          \(DEST[i+31:i] \leftarrow \) RoundFPControl_MXCSR\((SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])\)
          FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *\(DEST[i+31:i]\) remains unchanged* 
            ELSE ; zeroing-masking
              \(DEST[i+31:i] \leftarrow 0\)
            FI 
        FI;
    ENDFOR

\(DEST[\text{MAX}_VL-1:VL] \leftarrow 0\)

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

\(\text{VFMADDxxxPS} \_\_m512 \_\_mm512 \_\_mm_mask_fmadd_ps(\_\_m512 a, \_\_m512 b, \_\_m512 c);\)
\(\text{VFMADDxxxPS} \_\_m512 \_\_mm512 \_\_mm_maskz_fmadd_ps(\_\_m512 a, \_\_m512 b, \_\_m512 c, \_\_mmask16 k, \_\_mm512 a, \_\_m512 b, \_\_m512 c, \_\_mmask16 k);\)
\(\text{VFMADDxxxPS} \_\_m512 \_\_mm512 \_\_mm_mask3_fmadd_ps(\_\_m512 a, \_\_m512 b, \_\_m512 c, \_\_mmask16 k);\)
\(\text{VFMADDxxxPS} \_\_m512 \_\_mm512 \_\_mm_maskz_round_ps(\_\_m512 a, \_\_m512 b, \_\_m512 c, \_\_mmask16 k, \_\_mm512 a, \_\_m512 b, \_\_m512 c, \_\_mmask16 k, \_\_int r);\)
\(\text{VFMADDxxxPS} \_\_m512 \_\_mm512 \_\_mm_maskz3_round_ps(\_\_m512 a, \_\_m512 b, \_\_m512 c, \_\_mmask16 k, \_\_mm512 a, \_\_m512 b, \_\_m512 c, \_\_mmask16 k, \_\_int r);\)
\(\text{VFMADDxxxPS} \_\_m256 \_\_mm256 \_\_mm_mask_fmadd_ps(\_\_m256 a, \_\_m256 b, \_\_m256 c);\)
\(\text{VFMADDxxxPS} \_\_m256 \_\_mm256 \_\_mm_maskz_fmadd_ps(\_\_m256 a, \_\_m256 b, \_\_m256 c, \_\_mmask8 k, \_\_m256 a, \_\_m256 b, \_\_m256 c, \_\_mmask8 k);\)
\(\text{VFMADDxxxPS} \_\_m128 \_\_mm128 \_\_mm_mask_fmadd_ps(\_\_m128 a, \_\_m128 b, \_\_m128 c, \_\_mmask8 k, \_\_m128 a, \_\_m128 b, \_\_m128 c);\)
\(\text{VFMADDxxxPS} \_\_m128 \_\_mm128 \_\_mm_maskz_fmadd_ps(\_\_m128 a, \_\_m128 b, \_\_m128 c, \_\_mmask8 k);\)
\(\text{VFMADDxxxPS} \_\_m256 \_\_mm256 \_\_mm_mask3_fmadd_ps(\_\_m256 a, \_\_m256 b, \_\_m256 c, \_\_mmask8 k);\)
\(\text{VFMADDxxxPS} \_\_m128 \_\_mm128 \_\_mm_mask3_fmadd_ps(\_\_m128 a, \_\_m128 b, \_\_m128 c);\)
\(\text{VFMADDxxxPS} \_\_m256 \_\_mm256 \_\_mm_mask3_round_ps(\_\_m256 a, \_\_m256 b, \_\_m256 c, \_\_mmask8 k);\)
\(\text{VFMADDxxxPS} \_\_m128 \_\_mm128 \_\_mm_mask3_round_ps(\_\_m128 a, \_\_m128 b, \_\_m128 c);\)

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1, xmm2, xmm3/m64</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1, xmm2, xmm3/m64</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1, xmm2, xmm3/m64</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1 (k1)[z], xmm2, xmm3/m64(er)</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 (k1)[z], xmm2, xmm3/m64(er)</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 (k1)[z], xmm2, xmm3/m64(er)</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>T1S</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD multiply-add computation on the low double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

VFMADD132SD: Multiplies the low double-precision floating-point value from the first source operand to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point values in the second source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low double-precision floating-point value from the second source operand to the low double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low double-precision floating-point value from the second source to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

### VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

If (EVEX.b = 1) and SRC3 *is a register*

- THEN
  - SET_RM(EVEX.RC);
  - ELSE
    - SET_RM(MXCSR.RM);

- FI;

- IF k1[0] or *no writemask*
  - THEN DEST[63:0] ← RoundFPControl(DEST[63:0] * SRC3[63:0] + SRC2[63:0])
  - ELSE
    - IF *merging-masking* ; merging-masking
      - THEN *DEST[63:0] remains unchanged*
    - ELSE ; zeroing-masking
      - THEN DEST[63:0] ← 0

- FI;

- FI;

- DEST[127:64] ← DEST[127:64]
- DEST[MAX_VL-1:128] ← 0

### VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

If (EVEX.b = 1) and SRC3 *is a register*

- THEN
  - SET_RM(EVEX.RC);
  - ELSE
    - SET_RM(MXCSR.RM);

- FI;

- IF k1[0] or *no writemask*
  - THEN DEST[63:0] ← RoundFPControl(SRC2[63:0] * DEST[63:0] + SRC3[63:0])
  - ELSE
    - IF *merging-masking* ; merging-masking
      - THEN *DEST[63:0] remains unchanged*
    - ELSE ; zeroing-masking
      - THEN DEST[63:0] ← 0

- FI;

- FI;

- DEST[127:64] ← DEST[127:64]
- DEST[MAX_VL-1:128] ← 0
VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
                FI;
    FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

DEST[63:0] ← MAX_VL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-1:128] ← 0

VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-1:128] ← 0

VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)

DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxSD __m128d _mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask_fmadd_round_sd(__m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d _mm_mask_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, __mmask8 k);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DS.LG.66.0F38.01 99 / r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DS.LG.66.0F38.01 A9 / r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DS.LG.66.0F38.01 B9 / r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DS.LG.66.0F38.01 99 / r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DS.LG.66.0F38.01 A9 / r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DS.LG.66.0F38.01 B9 / r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>RVM</td>
</tr>
<tr>
<td>T1S</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD multiply-add computation on single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

VFMADD132SS: Multiplies the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN DEST[31:0] \( \leftarrow \) RoundFPControl(DEST[31:0] * SRC3[31:0] + SRC2[31:0])
   ELSE
       IF *merging-masking* ; merging-masking
           THEN *DEST[31:0] remains unchanged*
           ELSE ; zeroing-masking
               THEN DEST[31:0] \( \leftarrow \) 0
       FI;
   FI;
DEST[127:32] \( \leftarrow \) DEST[127:32]
DEST[MAX_VL-1:128] \( \leftarrow \) 0

**VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN DEST[31:0] \( \leftarrow \) RoundFPControl(SRC2[31:0] * DEST[31:0] + SRC3[31:0])
   ELSE
       IF *merging-masking* ; merging-masking
           THEN *DEST[31:0] remains unchanged*
           ELSE ; zeroing-masking
               THEN DEST[31:0] \( \leftarrow \) 0
       FI;
   FI;
DEST[127:32] \( \leftarrow \) DEST[127:32]
DEST[MAX_VL-1:128] \( \leftarrow \) 0
VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
   ELSE
       IF *merging-masking* ; merging-masking
           THEN *DEST[31:0]] remains unchanged*
           ELSE ; zeroing-masking
               THEN DEST[31:0] ← 0
       FI;
   FI;
DEST[MAX_VL-1:128] ← 0

VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[MAX_VL-1:128] ← 0

VFMADD123SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[MAX_VL-1:128] ← 0

VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMADDxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxSS __m128 _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxSS __m128 _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxSS __m128 _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxSS __m128 _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxSS __m128 _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxSS __m128 _mm_maskz_fmadd_round_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
### VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>
**VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD**—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

**VFMADDSUB132PD**: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB213PD**: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMADDSUB231PD**: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions**: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version**: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.
VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “-“ symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PD DEST, SRC2, SRC3
IF (VEX.128) THEN
  DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
  DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
  DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
  DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
  DEST[127:64] ← RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
FI

VFMADDSUB213PD DEST, SRC2, SRC3
IF (VEX.128) THEN
  DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
  DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
  DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
  DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
  DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
FI

VFMADDSUB231PD DEST, SRC2, SRC3
IF (VEX.128) THEN
  DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
  DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
  DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
  DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
  DEST[127:64] ← RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
FI
VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    DEST[i+63:i] ← RoundFPControl(DEST[i+63:i] * SRC3[i+63:i] - SRC2[i+63:i])
                    ELSE
                        DEST[i+63:i] ← RoundFPControl(DEST[i+63:i] * SRC3[i+63:i] + SRC2[i+63:i])
                FI
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+63:i] ← 0
                    FI
            FI
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
              RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
            ELSE
              DEST[i+63:i] ←
              RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
              RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
              DEST[i+63:i] ←
              RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
          FI;
      FI
    ELSE
      *merging-masking*  ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE  ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
  FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ← RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                ELSE DEST[i+63:i] ← RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR

DEST[MAX_VL-1:VL] ← 0
VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
            ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
            ELSE
                DEST[i+63:i] ←
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI;
          FI
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
          FI
      FI
  ELSE
  ENDFOR
  DEST[MAX_VL-1:VL] ← 0
VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR

DEST[MAX_VL-1:VL] ← 0
VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i]
                            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                            ELSE
                                DEST[i+63:i]
                                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                                FI;
                        ELSE
                            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                        FI;
                ELSE
                    THEN
                        DEST[i+63:i]
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                    ELSE
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                    FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *
                        DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
            FI
        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

VFMADDSUBxxxPD __m512d __mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m256d __mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_mask_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_fmadsub_pd(__m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m256d __mm256_fmadsub_pd(__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
### VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W0 96 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from <code>xmm1</code> and <code>xmm3/mem</code>, add/subtract elements in <code>xmm2</code> and put result in <code>xmm1</code>.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 A6 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from <code>xmm1</code> and <code>xmm2</code>, add/subtract elements in <code>xmm3/mem</code> and put result in <code>xmm1</code>.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 B6 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from <code>xmm2</code> and <code>xmm3/mem</code>, add/subtract elements in <code>xmm1</code> and put result in <code>xmm1</code>.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 96 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from <code>ymm1</code> and <code>ymm3/mem</code>, add/subtract elements in <code>ymm2</code> and put result in <code>ymm1</code>.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 A6 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from <code>ymm2</code> and <code>ymm3/mem</code>, add/subtract elements in <code>ymm1</code> and put result in <code>ymm1</code>.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 B6 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from <code>ymm2</code> and <code>ymm3/mem</code>, add/subtract elements in <code>ymm1</code> and put result in <code>ymm1</code>.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 A6 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from <code>xmm1</code> and <code>xmm2</code>, add/subtract elements in <code>xmm3/m128/m32bcst</code> and put result in <code>xmm1</code> subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 B6 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from <code>xmm2</code> and <code>xmm3/m128/m32bcst</code>, add/subtract elements in <code>xmm1</code> and put result in <code>xmm1</code> subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 A6 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from <code>ymm1</code> and <code>ymm2</code>, add/subtract elements in <code>ymm3/m256/m32bcst</code> and put result in <code>ymm1</code> subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 B6 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from <code>ymm2</code> and <code>ymm3/m256/m32bcst</code>, add/subtract elements in <code>ymm1</code> and put result in <code>ymm1</code> subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 A6 /r</td>
<td>FV V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from <code>zmm1</code> and <code>zmm2</code>, add/subtract elements in <code>zmm3/m512/m32bcst</code> and put result in <code>zmm1</code> subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 B6 /r</td>
<td>FV V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from <code>zmm2</code> and <code>zmm3/m512/m32bcst</code>, add/subtract elements in <code>zmm1</code> and put result in <code>zmm1</code> subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

VFMADDSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

EVR encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.
**Operation**

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADDSUB132PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI

For i = 0 to MAXNUM -1{
   n ← 64*i;
}

IF (VEX.128) THEN
   DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAX_VL-1:256] ← 0
FI

**VFMADDSUB213PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI

For i = 0 to MAXNUM -1{
   n ← 64*i;
}

IF (VEX.128) THEN
   DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAX_VL-1:256] ← 0
FI
VFMADDSUB231PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) (4, 128), (8, 256),= (16, 512)
IF (VL = 512) AND (EVEX.b = 1) THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF j *is even*
            THEN DEST[i+31:i] ← RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            ELSE DEST[i+31:i] ← RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
        FI
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
              RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
            ELSE
              DEST[i+31:i] ←
              RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
              RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
            ELSE
              DEST[i+31:i] ←
              RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
          FI;
      ELSE
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
          FI
      FI
  FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
                        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                ELSE
                    DEST[i+31:i] ←
                        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                FI;
            ELSE
                IF (EVEX.b = 1)
                    THEN
                        DEST[i+31:i] ←
                            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                    ELSE
                        DEST[i+31:i] ←
                            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                    FI;
            FI;
ENDFOR
IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
    DEST[i+31:i] ← 0
FI

ELSE;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

**VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
                FI
            FI
        ENDIF
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                    FI;
                ELSE
                    IF *merging-masking* ; merging-masking
                        THEN *DEST[i+31:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+31:i] ← 0
                    FI
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        ENDIF
    ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUBxxxPS __m512 __m512_fmadsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __m512_fmadsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __m512_mask_fmadsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __m512_maskz_fmadsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __m512_mask3_fmadsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512 __m512_mask_fmadsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __m512_maskz_fmadsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m256 __m256_mask_fmadsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __m256_maskz_fmadsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __m256_mask3_fmadsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128 __m128_mask_fmadsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __m128_maskz_fmadsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __m128_mask3_fmadsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m256 __m256_fmadsub_ps(__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
VFMSUBADD132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

EVA encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with writemask k1.
VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMSUBADD132PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

\[
\begin{align*}
\text{DEST}[63:0] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{DEST}[63:0] \times \text{SRC3}[63:0] + \text{SRC2}[63:0]) \\
\text{DEST}[127:64] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{DEST}[127:64] \times \text{SRC3}[127:64] - \text{SRC2}[127:64]) \\
\text{DEST}[\text{MAX}_\text{VL}:1:128] &\Leftarrow 0
\end{align*}
\]

ELSEIF (VEX.256)

\[
\begin{align*}
\text{DEST}[63:0] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{DEST}[63:0] \times \text{SRC3}[63:0] + \text{SRC2}[63:0]) \\
\text{DEST}[127:64] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{DEST}[127:64] \times \text{SRC3}[127:64] - \text{SRC2}[127:64]) \\
\text{DEST}[191:128] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{DEST}[191:128] \times \text{SRC3}[191:128] + \text{SRC2}[191:128]) \\
\text{DEST}[255:192] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{DEST}[255:192] \times \text{SRC3}[255:192] - \text{SRC2}[255:192])
\end{align*}
\]

FI

**VFMSUBADD213PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

\[
\begin{align*}
\text{DEST}[63:0] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[63:0] \times \text{DEST}[63:0] + \text{SRC3}[63:0]) \\
\text{DEST}[127:64] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[127:64] \times \text{SRC3}[127:64] - \text{DEST}[127:64]) \\
\text{DEST}[\text{MAX}_\text{VL}:1:128] &\Leftarrow 0
\end{align*}
\]

ELSEIF (VEX.256)

\[
\begin{align*}
\text{DEST}[63:0] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[63:0] \times \text{SRC3}[63:0] + \text{DEST}[63:0]) \\
\text{DEST}[127:64] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[127:64] \times \text{SRC3}[127:64] - \text{DEST}[127:64]) \\
\text{DEST}[191:128] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[191:128] \times \text{SRC3}[191:128] + \text{DEST}[191:128]) \\
\text{DEST}[255:192] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[255:192] \times \text{SRC3}[255:192] - \text{DEST}[255:192])
\end{align*}
\]

FI

**VFMSUBADD231PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

\[
\begin{align*}
\text{DEST}[63:0] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[63:0] \times \text{SRC3}[63:0] + \text{DEST}[63:0]) \\
\text{DEST}[127:64] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[127:64] \times \text{SRC3}[127:64] - \text{DEST}[127:64]) \\
\text{DEST}[\text{MAX}_\text{VL}:1:128] &\Leftarrow 0
\end{align*}
\]

ELSEIF (VEX.256)

\[
\begin{align*}
\text{DEST}[63:0] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[63:0] \times \text{SRC3}[63:0] + \text{DEST}[63:0]) \\
\text{DEST}[127:64] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[127:64] \times \text{SRC3}[127:64] - \text{DEST}[127:64]) \\
\text{DEST}[191:128] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[191:128] \times \text{SRC3}[191:128] + \text{DEST}[191:128]) \\
\text{DEST}[255:192] &\Leftarrow \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[255:192] \times \text{SRC3}[255:192] - \text{DEST}[255:192])
\end{align*}
\]

FI
VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

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

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
  FI;

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
        IF j *is even*
            THEN DEST[i+63:i] ←
                RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            ELSE DEST[i+63:i] ←
                RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        FI
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI
ENDFOR

DEST[MAX_VL-1:VL] ← 0
VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←
              RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
          ELSE
              DEST[i+63:i] ←
              RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
        FI;
      ELSE
        IF (EVEX.b = 1)
          THEN
            DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        ELSE
            DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN DEST[i+63:i] ← RoundFPControl(SRC2[i+63:i] * DEST[i+63:i] + SRC3[i+63:i])
        ELSE DEST[i+63:i] ← RoundFPControl(SRC2[i+63:i] * DEST[i+63:i] - SRC3[i+63:i])
      FI
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI
  END FOR

DEST[MAX_VL-1:VL] ← 0
VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                                DEST[i+63:i] ←
                                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                            ELSE
                                DEST[i+63:i] ←
                                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                            FI;
                        ELSE
                                IF (EVEX.b = 1)
                                    THEN
                                            DEST[i+63:i] ←
                                            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                                        ELSE
                                            DEST[i+63:i] ←
                                            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                                        FI;
                                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
VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] ←  
              RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
            ELSE
              DEST[i+63:i] ← 
              RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                DEST[i+63:i] ← 
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
              ELSE
                DEST[i+63:i] ← 
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
              FI;
            ELSE
              IF *merging-masking* ; merging-masking
              THEN *DEST[i+63:i] remains unchanged*
              ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
              FI
            FI
        FI
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBADDxxxPD __m512d _mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d _mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m256d _mm256_fmsubadd_pd(__m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d _mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d _mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m128d _mm128_mask_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d _mm128_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d _mm128_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d _mm128_fmsubadd_pd(__m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d _mm128_mask_fmsubadd_round_pd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBADDxxxPD __m128d _mm128_maskz_fmsubadd_round_pd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W0 97 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128, subtract/add elements in xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 A7 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 B7 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 97 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256, subtract/add elements in ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 A7 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 B7 /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 97 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 A7 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 B7 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 97 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 A7 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 B7 /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 97 /r</td>
<td>FV V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 A7 /r</td>
<td>FV V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 B7 /r</td>
<td>FV V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

**Description**
VFMSUBADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version:** The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.
**Operation**

In the operations below, "**" and "*" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMSUBADD132PS Destination, Source 2, Source 3**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSE IF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM -1{
   n ← 64*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}

IF (VEX.128) THEN
   DEST[MAX_VL-1:128] ← 0
ELSE IF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

**VFMSUBADD213PS Destination, Source 2, Source 3**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSE IF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM -1{
   n ← 64*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}

IF (VEX.128) THEN
   DEST[MAX_VL-1:128] ← 0
ELSE IF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

**VFMSUBADD231PS Destination, Source 2, Source 3**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSE IF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM -1{
   n ← 64*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}

IF (VEX.128) THEN
   DEST[MAX_VL-1:128] ← 0
ELSE IF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI
    ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
  \(i \leftarrow j \ast 32\)
  IF \(k1[j] \text{ OR } \text{no writemask}\)
    THEN
      IF \(j \text{ is even}\)
        THEN
          IF \(\text{EVEX.b = 1}\)
            THEN
              DEST\([i+31:i]\) \leftarrow \text{RoundFPControl_MXCSR}(\text{DEST}\([i+31:i]\) \ast \text{SRC3}\([31:0]\) + \text{SRC2}\([i+31:i]\))
            ELSE
              DEST\([i+31:i]\) \leftarrow \text{RoundFPControl_MXCSR}(\text{DEST}\([i+31:i]\) \ast \text{SRC3}\([i+31:i]\) + \text{SRC2}\([i+31:i]\))
            FI;
          ELSE
            IF \(\text{EVEX.b = 1}\)
              THEN
                DEST\([i+31:i]\) \leftarrow \text{RoundFPControl_MXCSR}(\text{DEST}\([i+31:i]\) \ast \text{SRC3}\([31:0]\) - \text{SRC2}\([i+31:i]\))
              ELSE
                DEST\([i+31:i]\) \leftarrow \text{RoundFPControl_MXCSR}(\text{DEST}\([i+31:i]\) \ast \text{SRC3}\([i+31:i]\) - \text{SRC2}\([i+31:i]\))
              FI;
            ELSE
              IF \(\text{merging-masking}\)
                THEN \(\text{DEST}[i+31:i]\) \text{ remains unchanged}\)
              ELSE \(\text{zeroing-masking}\)
                DEST\([i+31:i]\) \leftarrow 0
              FI;
          FI;
      FI;
    ELSE
      IF \(\text{merging-masking}\) ; merging-masking
        THEN \(\text{DEST}[i+31:i]\) \text{ remains unchanged}\)
      ELSE ; zeroing-masking
        DEST\([i+31:i]\) \leftarrow 0
      FI;
    FI;
ENDFOR
DEST[\text{MAX\_VL-1:VL}] \leftarrow 0
VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
   THEN
      SET_RM(EVEX.RC);
   ELSE
      SET_RM(MXCSR.RM);
   FI;

FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF j *is even*
            THEN DEST[i+31:i] ←
               RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            ELSE DEST[i+31:i] ←
               RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
         FI
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
         ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
         FI
      FI
ENDFOR

DEST[MAX_VL-1:VL] ← 0
VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                        ELSE
                            DEST[i+31:i] ←
                            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                        FI;
                    ELSE
                        IF (EVEX.b = 1)
                            THEN
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                            ELSE
                                DEST[i+31:i] ←
                                RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                            FI;
                        ELSE
                            IF *merging-masking* ; merging-masking
                                THEN *DEST[i+31:i] remains unchanged*;
                            ELSE ; zeroing-masking
                                DEST[i+31:i] ← 0
                            FI
                        FI
        ELSE
            ENDFOR
    DEST[MAX_VL-1:VL] ← 0
VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF j *is even*
    THEN
      DEST[i+31:i] ←
      RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
    ELSE
      DEST[i+31:i] ←
      RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
    FI
  ELSE
    IF *merging-masking* ; merging-masking
    THEN
      *DEST[i+31:i] remains unchanged*
    ELSE
      ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI
ENDFOR

DEST[MAX_VL-1:VL] ← 0
VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1  
i ← j * 32  
IF k1[j] OR *no writemask*  
    THEN  
        IF j *is even*  
            THEN  
                IF (EVEX.b = 1)  
                    THEN  
                        DEST[i+31:i] ←  
                            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])  
                    ELSE  
                        DEST[i+31:i] ←  
                            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])  
                    FI;  
                ELSE  
                    IF (EVEX.b = 1)  
                        THEN  
                            DEST[i+31:i] ←  
                                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])  
                        ELSE  
                            DEST[i+31:i] ←  
                                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])  
                        FI;  
                FI  
            ELSE  
                IF *merging-masking* ; merging-masking  
                    THEN *DEST[i+31:i] remains unchanged*  
                ELSE ; zeroing-masking  
                    DEST[i+31:i] ← 0  
                FI  
        FI  
    FI  
ENDFOR  
DEST[MAX_VL-1:VL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUBADDxxxPS __m512 __m512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __m512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __m512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __m512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __m512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512 __m512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __m512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __m512_mask3_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBADDxxxPS __m256 __m256_fmsubadd_ps(__m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 __m256_fmsubadd_round_ps(__m256 a, __m256 b, __m256 c, int r);
VFMSUBADDxxxPS __m256 __m256_mask_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBADDxxxPS __m256 __m256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m512 __m512_mask_fmsubadd_ps(__m512 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m512 __m512_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m512 __m512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask8 k);
VFMSUBADDxxxPS __m512 __m512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m256 __m256_fmsubadd_ps(__m256 a, __m256 b, __m256 c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
## VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W1 9A /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128, subtract xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 AA /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 BA /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 9A /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 AA /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 BA /r</td>
<td>RVM V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 9A /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 AA /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 BA /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract xmm1 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 9A /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 AA /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 BA /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract ymm1 and put result in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 9A /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract zmm2 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 AA /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 BA /r</td>
<td>FV V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMSUB132PD**:
Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PD**:
Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PD**:
Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVE encoded versions**: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version**: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version**: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PD DEST, SRC2, SRC3 (VEX encoded versions)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(Dest[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

VFMSUB213PD DEST, SRC2, SRC3 (VEX encoded versions)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

VFMSUB231PD DEST, SRC2, SRC3 (VEX encoded versions)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
                        RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] ←
                        RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] ←
  RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
  ELSE
  IF *merging-masking* ; merging-masking
  THEN *DEST[i+63:i] remains unchanged*
  ELSE ; zeroing-masking
  DEST[i+63:i] ← 0
  FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
   i ← j * 64
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1)
            THEN
               DEST[i+63:i] ←
               RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0]) +31:i)
            ELSE
               DEST[i+63:i] ←
               RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI;
         ELSE
            IF *merging-masking* ; merging-masking
               THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+63:i] ← 0
            FI
         FI
   FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
   i ← j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] ←
      RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
         ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
         FI
   FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(SRC2[i+63:j]*SRC3[63:0] - DEST[i+63:j])
          ELSE
            DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:512] - DEST[i+63:512])
        FI;
      ELSE
        IF *merging-masking*
          THEN *DEST[i+63:j] remains unchanged*
        ELSE
          DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxPD __m512d _mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_pd(__mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_round_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_round_pd(__mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask_fmsub_pd (__m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_pd (__mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_pd (__mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_mask_fmsub_pd (__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_maskz_fmsub_pd (__mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_mask3_fmsub_pd (__mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_fmsub_round_pd (__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxPD __m128d _mm_mask_fmsub_round_pd (__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_maskz_fmsub_round_pd (__mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxPD __m128d _mm_mask3_fmsub_round_pd (__mmask8 k, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/E</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 BA /r VFMSUB231PS ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed-single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed-single-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed-single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst{er}</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed-single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst{er}</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed-single-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 BA /r VFMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst{er}</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed-single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Description
Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
**Operation**

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN

```
MAXNUM ← 2
```
ELSEIF (VEX.256)

```
MAXNUM ← 4
```
FI

For i = 0 to MAXNUM-1 {

```
n ← 32*i;
DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
```
}

IF (VEX.128) THEN

```
DEST[MAX_VL-1:128] ← 0
```
ELSEIF (VEX.256)

```
DEST[MAX_VL-1:256] ← 0
```
FI

**VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN

```
MAXNUM ← 2
```
ELSEIF (VEX.256)

```
MAXNUM ← 4
```
FI

For i = 0 to MAXNUM-1 {

```
n ← 32*i;
DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
```
}

IF (VEX.128) THEN

```
DEST[MAX_VL-1:128] ← 0
```
ELSEIF (VEX.256)

```
DEST[MAX_VL-1:256] ← 0
```
FI

**VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN

```
MAXNUM ← 2
```
ELSEIF (VEX.256)

```
MAXNUM ← 4
```
FI

For i = 0 to MAXNUM-1 {

```
n ← 32*i;
DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
```
}

IF (VEX.128) THEN

```
DEST[MAX_VL-1:128] ← 0
```
ELSEIF (VEX.256)

```
DEST[MAX_VL-1:256] ← 0
```
FI
VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl([DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i]]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] ←
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0
VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
            ELSE
                DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI;
         ELSE
            IF *merging-masking* ; merging-masking
               THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+31:i] ← 0
            FI
      FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
   THEN
      SET_RM(EVEX.RC);
   ELSE
      SET_RM(MXCSR.RM);
   FI;
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ←
          RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
         ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
         FI
   FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                ELSE
                    DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                    FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] ← 0
                    FI
            FI
    ENDFOR

DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxPS __m512 _mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_fmsub_round_ps (__m256 a, __m256 b, __m256 c, int r);
VFMSUBxxxPS __m256 _mm256_mask_fmsub_ps (__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_mask_fmsub_round_ps (__m256 a, __mmask8 k, __m256 b, __m256 c, int r);
VFMSUBxxxPS __m256 _mm256_maskz_fmsub_ps (__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_mask3_fmsub_ps (__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 _mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_mask_fmsub_round_ps(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxPS __m128 _mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 9B /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VFMSUB132SD xmm1, xmm2, xmm3/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 AB /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>VFMSUB213SD xmm1, xmm2, xmm3/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 BB /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VFMSUB231SD xmm1, xmm2, xmm3/m64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 9B /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VFMSUB132SD xmm1 {k1}[z], xmm2, xmm3/m64[er]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 AB /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>VFMSUB213SD xmm1 {k1}[z], xmm2, xmm3/m64[er]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 BB /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VFMSUB231SD xmm1 {k1}[z], xmm2, xmm3/m64[er]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>T1S</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

**VFMSUB132SD**

Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMSUB213SD**

Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMSUB231SD**

Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version**

The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low quadword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
THEN  DEST[63:0] \leftarrow \text{RoundFPControl}(\text{DEST}[63:0]*\text{SRC3}[63:0] - \text{SRC2}[63:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] \leftarrow 0
    FI;
FI;
DEST[127:64] \leftarrow DEST[127:64]
DEST[MAX_VL-1:128] \leftarrow 0

**VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
THEN  DEST[63:0] \leftarrow \text{RoundFPControl}(\text{SRC2}[63:0]*\text{DEST}[63:0] - \text{SRC3}[63:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] \leftarrow 0
    FI;
FI;
DEST[127:64] \leftarrow DEST[127:64]
DEST[MAX_VL-1:128] \leftarrow 0
VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] ← 0
        FI;
    FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxSD __m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d _mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 9B /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 AB /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 BB /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 9B /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 AB /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 BB /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM[reg (r, w)]</td>
<td>VEX.vvvv (r)</td>
<td>ModRM[x/m (r)]</td>
<td>NA</td>
</tr>
<tr>
<td>T1S</td>
<td>ModRM[reg (r, w)]</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM[x/m (r)]</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

VFMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation
In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN  DEST[31:0] \leftarrow RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[31:0] remains unchanged*
      ELSE ; zeroing-masking
         THEN DEST[31:0] \leftarrow 0
      FI;
   FI;
DEST[127:32] \leftarrow DEST[127:32]
DEST[MAX_VL-1:128] \leftarrow 0

VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_RM(EVEX.RC);
   ELSE
       SET_RM(MXCSR.RM);
   FI;
IF k1[0] or *no writemask*
   THEN  DEST[31:0] \leftarrow RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[31:0] remains unchanged*
      ELSE ; zeroing-masking
         THEN DEST[31:0] \leftarrow 0
      FI;
   FI;
DEST[127:32] \leftarrow DEST[127:32]
DEST[MAX_VL-1:128] \leftarrow 0
VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[MAX_VL-1:128] ← 0

VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(DEST[31:0]*SRC2[31:0] - SRC2[31:0])
DEST[MAX_VL-1:128] ← 0

VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[MAX_VL-1:128] ← 0

VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask2_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask2_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEV-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
### VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm0{k1}{z}, xmm1, xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m64bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m64bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 9C /r VFNMADD132PD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 AC /r VFNMADD213PD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m64bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 BC /r VFNMADD231PD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Description
VFNMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).
VFNMADD213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).
VFNMADD231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).
EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.
VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.
VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation
In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).
VFNMADD132PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

VFNMADD213PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

VFNMADD231PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] ←
    RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
      THEN
        DEST[i+63:i] ←
        RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) + SRC2[i+63:i])
      ELSE
        DEST[i+63:i] ←
        RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR

DEST[MAX_VL-1:VL] ← 0

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[63:0])
                ELSE
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:3])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
        FI;
    ENDFOR

DEST[MAX_VL-1:VL] ← 0
VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] ← RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ← RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] ← RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_round_pd (__m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxPD __m256d _mm256_fnmadd_round_pd (__m256d a, __m256d b, __m256d c, int r);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_round_pd (__m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxPD __m256d _mm256_fnmadd_round_pd (__m256d a, __m256d b, __m256d c, int r);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 BC /r VFNMADD231PS ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.0 BC /r VFNMADD231PS ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m32bcst and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm2 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m32bcst and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 BC /r VFNMADD231PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm1 and put result in ymm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 9C /r VFNMADD132PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm2 and put result in zmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 AC /r VFNMADD213PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m32bcst and put result in zmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 BC /r VFNMADD231PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>FV V/V AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

### Description
VFNMADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

ESEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

### Operation
In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFNMADD132PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN
   MAXNUM ← 2
ELSEIF (VEX.256)
   MAXNUM ← 4
FI

FOR i = 0 to MAXNUM-1 {
   n ← 32*i;
   DEST[n+31:n] ← RoundFPControl_MXCSR(-(DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}

IF (VEX.128) THEN
   DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
   DEST[MAX_VL-1:256] ← 0
FI

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRMreg(r, w)</td>
<td>VEX.vvvv(r)</td>
<td>ModRMr/m(r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRMreg(r, w)</td>
<td>EVEX.vvvv(r)</td>
<td>ModRMr/m(r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
VFNMADD213PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(-(SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
VFNMADD231PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(-(SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) + SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
          FI
        FI;
    ENDFOR
  DEST[MAX_VL-1:VL] ← 0

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
    RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
        THEN
            DEST[i+31:i] ← RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])
        ELSE
            DEST[i+31:i] ← RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1)
            THEN
               DEST[i+31:i] ←
               RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
            ELSE
               DEST[i+31:i] ←
               RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
         FI;
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+31:i] ← 0
         FI
      FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxPS __m512 _mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMADDxxxPS __m256 _mm256_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_maskz_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_mask3_fnmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_mask3_fnmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 9D /r VFNMADD132SD xmm1, xmm2, xmm3/m64</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 AD /r VFNMADD213SD xmm1, xmm2, xmm3/m64</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 BD /r VFNMADD231SD xmm1, xmm2, xmm3/m64</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 9D /r VFNMADD132SD xmm1{[k1][z]}, xmm2, xmm3/m64[er]</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64[er], negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 AD /r VFNMADD213SD xmm1{[k1][z]}, xmm2, xmm3/m64[er]</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64[er] and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 BD /r VFNMADD231SD xmm1{[k1][z]}, xmm2, xmm3/m64[er]</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64[er], negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>T1S</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

VFNMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvv/EVEX.vvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
Operation
In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFNMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)**

If (EVEX.b = 1) and SRC3 *is a register*
THEN
   SET_RM(EVEX.RC);
ELSE
   SET_RM(MXCSR.RM);
FI;

IF k1[0] or *no writemask*
THEN DEST[63:0] \[=\] RoundFPControl(-(DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
ELSE
   IF *merging-masking* ; merging-masking
   THEN *DEST[63:0] remains unchanged*
   ELSE ; zeroing-masking
       THEN DEST[63:0] \[=\] 0
   FI;
FI;

DEST[127:64] \[=\] DEST[127:64]
DEST[MAX_VL-1:128] \[=\] 0

**VFNMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)**

If (EVEX.b = 1) and SRC3 *is a register*
THEN
   SET_RM(EVEX.RC);
ELSE
   SET_RM(MXCSR.RM);
FI;

IF k1[0] or *no writemask*
THEN DEST[63:0] \[=\] RoundFPControl(-(SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
ELSE
   IF *merging-masking* ; merging-masking
   THEN *DEST[63:0] remains unchanged*
   ELSE ; zeroing-masking
       THEN DEST[63:0] \[=\] 0
   FI;
FI;

DEST[127:64] \[=\] DEST[127:64]
DEST[MAX_VL-1:128] \[=\] 0
VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN 
        DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged* 
                ELSE ; zeroing-masking
                    THEN DEST[63:0] ← 0
                FI;
        FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFNMADD132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR( -(DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFNMADD213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR( -(SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFNMADD231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR( -(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxSD __m128d _mm_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| VEX.DDS.LIG.66.0F38.W0 9D /r  
VFNMADD132SS xmm1, xmm2, xmm3/m32 | RVM | V/V | FMA | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 AD /r  
VFNMADD213SS xmm1, xmm2, xmm3/m32 | RVM | V/V | FMA | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1. |
| VEX.DDS.LIG.66.0F38.W0 BD /r  
VFNMADD231SS xmm1, xmm2, xmm3/m32 | RVM | V/V | FMA | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 9D /r  
VFNMADD132SS xmm1[k1]{k1}, xmm2, xmm3/m32{er} | T1S | V/V | AVX512F | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 AD /r  
VFNMADD213SS xmm1[k1]{k1}, xmm2, xmm3/m32{er} | T1S | V/V | AVX512F | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1. |
| EVEX.DDS.LIG.66.0F38.W0 BD /r  
VFNMADD231SS xmm1[k1]{k1}, xmm2, xmm3/m32{er} | T1S | V/V | AVX512F | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1. |

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>T1S</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[31:0] ← RoundFPControl(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[MAX_VL-1:128] ← 0

VFNMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
    FI;
DEST[MAX_VL-1:128] ← 0
VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN
        DEST[31:0] <- RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] <- 0
        FI;
    FI;
DEST[MAX_VL-1:128] <- 0

VFNMADD132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] <- RoundFPControl_MXCSR(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[MAX_VL-1:128] <- 0

VFNMADD213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] <- RoundFPControl_MXCSR(-(SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[MAX_VL-1:128] <- 0

VFNMADD231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] <- RoundFPControl_MXCSR(-(SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[MAX_VL-1:128] <- 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxSS __m128 _mm_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxSS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
### VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W1 9E</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 AE</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 BE</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 9E</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 AE</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 BE</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 9E</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 AE</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 BE</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 9E</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 AE</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m64bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 BE</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 9E</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 AE</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m64bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 BE</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
**Description**

VFNMSUB132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

**Operation**

In the operations below, “*” and “−” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)**

```plaintext
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 [
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
]

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
```

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
            IF *merging-masking*
                THEN *DEST[i+63:i] remains unchanged*
            ELSE
                zerosing-masking
                DEST[i+63:i] ← 0
            FI
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) - SRC2[i+63:i])
        ELSE
          DEST[i+63:i] ←
          RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        FI;
      ELSE
        IF *merging-masking*
          THEN *DEST[i+63:i] remains unchanged*
        ELSE
          ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
    RoundFPControl(-SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
  ELSE
    IF *merging-masking*
      THEN *DEST[i+63:i] remains unchanged*
    ELSE
      ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j \leftarrow 0 \text{ TO KL-1}
\begin{align*}
i & \leftarrow j \times 64 \\
\text{IF } k1[j] \text{ OR } \text{*no writemask*} \\
\text{THEN} \\
\text{IF (EVEX.b = 1)} \\
\text{THEN} \\
& \text{DEST}[i+63:i] \leftarrow \\
& \text{RoundFPControl\_MXCSR(-(SRC2}[i+63:i]*DEST}[i+63:i]) - SRC3[i+63:i]) \\
& \text{ELSE} \\
& \text{DEST}[i+63:i] \leftarrow \\
& \text{RoundFPControl\_MXCSR(-(SRC2}[i+63:i]*DEST}[i+63:i]) - SRC3[i+63:i]) \\
& \text{FI}; \\
\text{ELSE} \\
& \text{IF \text{*merging-masking*} \quad \text{; merging-masking} \\
& \text{THEN } *\text{DEST}[i+63:i] \text{ remains unchanged*} \\
& \text{ELSE} \quad \text{; zeroing-masking} \\
& \text{DEST}[i+63:i] \leftarrow 0 \\
& \text{FI} \\
& \text{FI}; \\
\text{ENDFOR} \\
\text{DEST}[\text{MAX\_VL-1:VL}] \leftarrow 0
\end{align*}

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)

IF (VL = 512) \text{ AND (EVEX.b = 1)}
\text{THEN} \\
\text{SET\_RM(EVEX.RC);} \\
\text{ELSE} \\
\text{SET\_RM(MXCSR.RM);} \\
\text{FI}; \\
\text{FOR j \leftarrow 0 \text{ TO KL-1}} \\
\text{i \leftarrow j \times 64} \\
\text{IF k1[j] OR \text{*no writemask*} \\
\text{THEN } \text{DEST}[i+63:i] \leftarrow \\
& \text{RoundFPControl\_}(\text{SRC2}[i+63:i]\text{SRC3}[i+63:i]) - \text{DEST}[i+63:i]) \\
\text{ELSE} \\
& \text{IF \text{*merging-masking*} \quad \text{; merging-masking} \\
& \text{THEN } *\text{DEST}[i+63:i] \text{ remains unchanged*} \\
& \text{ELSE} \quad \text{; zeroing-masking} \\
& \text{DEST}[i+63:i] \leftarrow 0 \\
& \text{FI} \\
& \text{FI}; \\
\text{ENDFOR} \\
\text{DEST}[\text{MAX\_VL-1:VL}] \leftarrow 0
VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR
    DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBxxxPD __m512d _mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
**VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 BE /r VFNMSUB231PS ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result subtract add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result subtract add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 BE /r VFNMSUB231PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result subtract add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 9E /r VFNMSUB132PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst[er], negate the multiplication result and subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 AE /r VFNMSUB213PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m32bcst[er] and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 BE /r VFNMSUB231PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst[er], negate the multiplication result subtract add to zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
VFNMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)

IF (VEX.128) THEN
    MAXNUM ← 2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI

For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n] * SRC3[n+31:n]) - SRC2[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256] ← 0
FI
VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

IF (VEX.128) THEN
    MAXNUM  2
ELSEIF (VEX.256)
    MAXNUM  4
FI

For i = 0 to MAXNUM-1 {
    n  32*i;
    DEST[n+31:n]  RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128]  0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256]  0
FI

VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

IF (VEX.128) THEN
    MAXNUM  2
ELSEIF (VEX.256)
    MAXNUM  4
FI

For i = 0 to MAXNUM-1 {
    n  32*i;
    DEST[n+31:n]  RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAX_VL-1:128]  0
ELSEIF (VEX.256)
    DEST[MAX_VL-1:256]  0
FI

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI

FOR j  0 TO KL-1
    i  j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i]  RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i]  0
                FI
            FI
ENDFOR

DEST[MAX_VL-1:VL]  0
VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] ←
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        ELSE
          DEST[i+31:i] ←
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
          FI
        FI;
  ENDFOR

DEST[MAX_VL-1:VL] ← 0

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
    RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] ← 0
        FI
      FI;
  ENDFOR

DEST[MAX_VL-1:VL] ← 0
VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
                ELSE
                    DEST[i+31:i] ←
                    RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
        RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
THEN
  IF (EVEX.b = 1)
    THEN
      DEST[+31:i] ←
      RoundFPControl_MXCSR(-(SRC2[+31:i]*SRC3[0:31]) - DEST[+31:i])
    ELSE
      DEST[+31:i] ←
      RoundFPControl_MXCSR(-(SRC2[+31:i]*SRC3[0:31]) - DEST[+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBxxxPS __m512 _mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m256 _mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_fnmsub_ps(__m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m256 _mm256_fnmsub_round_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m256 _mm256_fnmsub_round_ps(__m256 a, __m256 b, __m256 c, __m256 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.
## VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 9F /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 AF /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W1 BF /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 9F /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 AF /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W1 BF /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>T1S</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

**VFNMSUB132SD**: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMSUB213SD**: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFNMSUB231SD**: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “*+*“ and “-“ symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMBSU132SD** DEST, SRC2, SRC3 (EVEX encoded version)

If (EVEX.b = 1) and SRC3 *is a register*

- THEN
  - SET_RM(EVEX.RC);
- ELSE
  - SET_RM(MXCSR.RM);
- FI;

- IF k1[0] or *no writemask*
  - THEN DEST[63:0]  RoundFPControl(-(DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
  - ELSE
    - IF *merging-masking* ; merging-masking
      - THEN *DEST[63:0] remains unchanged*
    - ELSE ; zeroing-masking
      - THEN DEST[63:0]  0
  - FI;
- FI;

- DEST[127:64]  DEST[127:64]
- DEST[MAX_VL-1:128]  0

**VFNMBSU213SD** DEST, SRC2, SRC3 (EVEX encoded version)

If (EVEX.b = 1) and SRC3 *is a register*

- THEN
  - SET_RM(EVEX.RC);
- ELSE
  - SET_RM(MXCSR.RM);
- FI;

- IF k1[0] or *no writemask*
  - THEN DEST[63:0]  RoundFPControl(-(SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
  - ELSE
    - IF *merging-masking* ; merging-masking
      - THEN *DEST[63:0] remains unchanged*
    - ELSE ; zeroing-masking
      - THEN DEST[63:0]  0
  - FI;
- FI;

- DEST[127:64]  DEST[127:64]
- DEST[MAX_VL-1:128]  0
VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
    FI;
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR( (DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR( (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] ← RoundFPControl_MXCSR( (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] ← DEST[127:64]
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBxxxSD __m128d _mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSUBxxxSD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 9F /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 andxmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 AF /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.LIG.66.0F38.W0 BF /r</td>
<td>RVM</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 9F /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 AF /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.DDS.LIG.66.0F38.W0 BF /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

Description

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.
Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

    SET_RM(EVEX.RC);

ELSE

    SET_RM(MXCSR.RM);

FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] ← RoundFPControl(-(DEST[31:0]·SRC3[31:0]) - SRC2[31:0])

ELSE

    IF *merging-masking* ; merging-masking

    THEN *DEST[31:0] remains unchanged*

    ELSE ; zeroing-masking

    THEN DEST[31:0] ← 0

FI;

FI;


DEST[MAX_VL-1:128] ← 0

**VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

    SET_RM(EVEX.RC);

ELSE

    SET_RM(MXCSR.RM);

FI;

IF k1[0] or *no writemask*

THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]·DEST[31:0]) - SRC3[31:0])

ELSE

    IF *merging-masking* ; merging-masking

    THEN *DEST[31:0] remains unchanged*

    ELSE ; zeroing-masking

    THEN DEST[31:0] ← 0

FI;

FI;


DEST[MAX_VL-1:128] ← 0
VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_RM(EVEX.RC);
ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
THEN  DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
ELSE
    IF *merging-masking* ; merging-masking
    THEN  *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[31:0] ← 0
    FI;
FI;
DEST[MAX_VL-1:128] ← 0

VFNMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR( - (SRC2[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[MAX_VL-1:128] ← 0

VFNMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR( - (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[MAX_VL-1:128] ← 0

VFNMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] ← RoundFPControl_MXCSR( - (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBXXXSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBXXXSS __m128 _mm_mask_fnmsub_ss(__m128 a, __m128 b, __m128 c);
VFNMSUBXXXSS __m128 _mm_maskz_fnmsub_ss(__m128 a, __m128 b, __m128 c);
VFNMSUBXXXSS __m128 _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c);
VFNMSUBXXXSS __m128 _mm_mask_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBXXXSS __m128 _mm_maskz_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBXXXSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBXXXSS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.
### VFPCLASSPD—Tests Types Of a Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 66 /r ib VFPCLASSPD k2 [k1], xmm2/m128/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 66 /r ib VFPCLASSPD k2 [k1], ymm2/m256/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 66 /r ib VFPCLASSPD k2 [k1], zmm2/m512/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The FPCLASSPD instruction checks the packed double precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

![Figure 5-13. Imm8 Byte Specifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS](image)

**Table 5-6. Classifier Operations for VFPCLASSPD/SD/PS/SS**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>QNaN</td>
<td>PosZero</td>
<td>NegZero</td>
<td>PosINF</td>
<td>NegINF</td>
<td>Denormal</td>
<td>Negative</td>
<td>SNaN</td>
</tr>
<tr>
<td>Classifier</td>
<td>Checks for QNaN</td>
<td>Checks for +0</td>
<td>Checks for -0</td>
<td>Checks for +INF</td>
<td>Checks for -INF</td>
<td>Checks for Denormal</td>
<td>Checks for Negative finite</td>
<td>Checks for SNaN</td>
</tr>
</tbody>
</table>

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

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

CheckFPClassDP (tsrc[63:0], imm8[7:0])

```c
//* Start checking the source operand for special type */
NegNum ← tsrc[63];
IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
IF (ExpAllZeros AND MXCSR.DAZ) Then
    MantAllZeros ← 1;
ELSIF (tsrc[51:0]=0h) Then
    MantAllZeros ← 1;
FI;
ZeroNumber ← ExpAllZeros AND MantAllZeros
SignalingBit ← tsrc[51];

sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
Pinf_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +Inf
Ninf_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res ← NegNum AND NOT(ExpAllZeros) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND sNaN_res ) OR (imm8[1] AND Pzero_res ) OR

Return bResult;
} //* end of CheckFPClassDP() */
```

VFCLASSPD (EVEX Encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] ← CheckFPClassDP(SRC1[63:0], imm8[7:0]);
                ELSE
                    DEST[j] ← CheckFPClassDP(SRC1[i+63:j], imm8[7:0]);
                FI;
            ELSE DEST[j] ← 0 ; zeroing-masking only
        FI;
    ENDFOR
    DEST[MAX_KL-1:KL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFCLASSPD __m128d_mm_class_pd(__m128d a, int c);
VFCLASSPD __m256d_mm256_class_pd(__m256d a, int c);
VFCLASSPD __m512d_mm512_class_pd(__m512d a, int c);
VFCLASSPD __mmask8_mm_class_mask_pd(__mmask8 m, __m512d a, int c);
VFCLASSPD __mmask8_mm256_class_mask_pd(__mmask8 m, __m256d a, int c);
VFCLASSPD __mmask8_mm512_class_mask_pd(__mmask8 m, __m512d a, int c);
VFCLASSPD __mmask8_mm4_class_mask_pd(__mmask8 m, __m128d a, int c);
VFCLASSPD __mmask8_mm_class_mask4_pd(__mmask8 m, __m128d a, int c);
VFCLASSPD __mmask8_mm_class_mask8_pd(__mmask8 m, __m128d a, int c);
VFCLASSPD __mmask8_mm_class_mask8_pd(__mmask8 m, __m128d a, int c);
VFCLASSPD __mmask8_mm_class_mask8_pd(__mmask8 m, __m128d a, int c);
VFCLASSPD __mmask8_mm_class_mask8_pd(__mmask8 m, __m128d a, int c);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4

#UD If EVEX.vvvv != 1111B.
VFCLASSPS—Tests Types Of a Packed Float32 Values

**Opcode/Instruction** | **Op/En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**
--- | --- | --- | --- | ---
EVEX.128.66.0F3A.W0 66 /r ib VFPCLASSPS k2 [k1], xmm2/m128/m32bcst, imm8 | FV | V/V | AVX512VL AVX512DQ | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

EVEX.256.66.0F3A.W0 66 /r ib VFPCLASSPS k2 [k1], ymm2/m256/m32bcst, imm8 | FV | V/V | AVX512VL AVX512DQ | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

EVEX.512.66.0F3A.W0 66 /r ib VFPCLASSPS k2 [k1], zmm2/m512/m32bcst, imm8 | FV | V/V | AVX512DQ | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The FPCLASSPS instruction checks the packed single-precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:16/8/4] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

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

**Operation**

CheckFPClassSP (tsrc[31:0], imm8[7:0])

```c
//** Start checking the source operand for special type **/
NegNum ← tsrc[31];
IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1; FI;
IF (ExpAllZeros AND MXCSR.DAZ) Then MantAllZeros ← 1; ELSIF (tsrc[22:0]=0h) Then MantAllZeros ← 1; FI;
Fl;
ZeroNumber= ExpAllZeros AND MantAllZeros
SignalingBit= tsrc[22];

sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
```
Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
Pinf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
Ninf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR

Return bResult;
} //* end of CheckSPClassSP() *//

VFPCCLASSPS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
i ← j * 32
IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b == 1) AND (SRC *is memory*)
      THEN
        DEST[j] ← CheckFPClassDP(SRC1[31:0], imm8[7:0]);
      ELSE
        DEST[j] ← CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
      FI;
    ELSE  DEST[j] ← 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

Intel/C++ Compiler Intrinsic Equivalent

VFPCCLASSPS __mmask16 _mm512_fpclass_ps_mask( __m512 a, int c);
VFPCCLASSPS __mmask16 _mm512_mask_fpclass_ps_mask( __mmask16 m, __m512 a, int c)
VFPCCLASSPS __mmask8 _mm256_fpclass_ps_mask( __m256 a, int c)
VFPCCLASSPS __mmask8 _mm256_mask_fpclass_ps_mask( __mmask8 m, __m256 a, int c)
VFPCCLASSPS __mmask8 _mm_fpclass_ps_mask( __m128 a, int c)
VFPCCLASSPS __mmask8 _mm_mask_fpclass_ps_mask( __mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4

#UD         If EVEX.vvvv != 1111B.
VFPCLASSSD—Tests Types Of a Scalar Float64 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.66.0F3A.W1 67 /r ib VFPCLASSSD k2 [k1], xmm2/m64, imm8</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The FPCLASSSD instruction checks the low double precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1:1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

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

**Operation**

CheckFPClassDP (tsrc[63:0], imm8[7:0]){

```
NegNum ← tsrc[63];
IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
IF (ExpAllZeros AND MXCSR.DAZ) Then
  MantAllZeros ← 1;
ELSIF (tsrc[51:0]=0h) Then
  MantAllZeros ← 1;
FI;
ZeroNumber ← ExpAllZeros AND MantAllZeros
SignalingBit ← tsrc[51];

sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
PInf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
NInf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

bResult = (imm8[0] AND sNaN_res) OR (imm8[1] AND Pzero_res) OR

Return bResult;
} // end of CheckFPClassDP() */
```
VFPCLASSSD (EVEX encoded version)
IF k1[0] OR *no writemask*
   THEN DEST[0] ←
       CheckFPClassDP(SRC1[63:0], imm8[7:0])
   ELSE DEST[0] ← 0 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VFPCLASSSD __m128d __mmask8 fpclass_s_mask( __m128d a, int c)
VFPCLASSSD __m128d __mmask8 fpclass_s_mask( __m128d a, int c)

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E6
#UD If EVEX.vvvv != 1111B.
VFPCLASSSS—Tests Types Of a Scalar Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LI.G66.0F3A.W0 67 /r VFPCLASSSS k2 [k1], xmm2/m32, imm8</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The FPCLASSSS instruction checks the low single-precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1:1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

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

CheckFPClassSP (tsrc[31:0], imm8[7:0])

    //** Start checking the source operand for special type */
    NegNum ← tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELIF (tsrc[22:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    Pinf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    Ninf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR

    Return bResult;
} //** end of CheckSPClassSP() */

VFPCLASSSS (EVEX encoded version)

IF k1[0] OR *no writemask*
    THEN DEST[0] ←
        CheckFPClassSP(SRC1[31:0], imm8[7:0])
    ELSE DEST[0] ← 0 ; zeroing-masking only
    Fl;
    DEST[MAX_KL-1:1] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSS __m8 _mm_fpclass_ss_mask(__m128 a, int c)
VFPCLASSSS __m8 _mm_mask_fpclass_ss_mask(__m8 m, __m128 a, int c)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E6

#UD If EVEX.vvvv != 1111B.
VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/3 2-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DS.128.66.0F38.W1 92 /r VGATHERDPD xmm1, vm32x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DS.128.66.0F38.W1 93 /r VGATHERQPD xmm1, vm64x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather double-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DS.256.66.0F38.W1 92 /r VGATHERDPD ymm1, vm32y, ymm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.DS.256.66.0F38.W1 93 /r VGATHERQPD ymm1, vm64y, ymm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather double-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg (R): VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The instruction conditionally loads up to 2 or 4 double-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double-precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.
VEX.128 version: The instruction will gather two double-precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four double-precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

• If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.

• The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.

• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

• Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

• This instruction does not perform AC checks, and so will never deliver an AC fault.

• This instruction will cause a #UD if the address size attribute is 16-bit.

• This instruction will cause a #UD if the memory operand is encoded without the SIB byte.

• This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.

• The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.
Operation

DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VGATHERDPD (VEX.128 version)
FOR j ← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j ← 0 to 1
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPD (VEX.128 version)
FOR j ← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j ← 0 to 1
    k ← j * 64;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
VGATHERQPD (VEX.256 version)
FOR j ← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j ← 0 to 3
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)

VGATHERDPD (VEX.256 version)
FOR j ← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j ← 0 to 3
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
ENDFOR
(non-masked elements of the mask register have the content of respective element cleared)
**Intel C/C++ Compiler Intrinsic Equivalent**

VGATHERDPD: __m128d _mm_i32gather_pd (double const * base, __m128i index, const int scale);

VGATHERDPD: __m128d _mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);

VGATHERDPD: __m256d _mm256_i32gather_pd (double const * base, __m128i index, const int scale);

VGATHERDPD: __m256d _mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int scale);

VGATHERQPD: __m128d _mm_i64gather_pd (double const * base, __m128i index, const int scale);

VGATHERQPD: __m128d _mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);

VGATHERQPD: __m256d _mm256_i64gather_pd (double const * base, __m256i index, const int scale);

VGATHERQPD: __m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type 12.
VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS xmm1, vm32x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 92 /r VGATHERDPS ymm1, vm32y, ymm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather single-precision FP values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64y, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather single-precision FP values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero’ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.
VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

• If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.

• The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.

• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

• Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

• This instruction does not perform AC checks, and so will never deliver an AC fault.

• This instruction will cause a #UD if the address size attribute is 16-bit.

• This instruction will cause a #UD if the memory operand is encoded without the SIB byte.

• This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.

• The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.
Operation

DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VGATHERDPS (VEX.128 version)
FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
MASK[VLMAX-1:128] ← 0;
FOR j ← 0 to 3
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPS (VEX.128 version)
FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
MASK[VLMAX-1:128] ← 0;
FOR j ← 0 to 1
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[VLMAX-1:64] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
VGATHERDPS (VEX.256 version)

FOR j ← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR

FOR j ← 0 to 7
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPS (VEX.256 version)

FOR j ← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR

FOR j ← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR

MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;

(non-masked elements of the mask register have the content of respective element cleared)
Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPS: __m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);
VGATHERDPS: __m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);
VGATHERDPS: __m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);
VGATHERDPS: __m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);
VGATHERQPS: __m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);
VGATHERQPS: __m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);
VGATHERQPS: __m256 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);
VGATHERQPS: __m256 _mm256_mask_i64gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 12
VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 92 /vsib VGATHERDPS xmm1 {k1}, vm32x</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 92 /vsib VGATHERDPS ymm1 {k1}, vm32y</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 92 /vsib VGATHERDPS zmm1 {k1}, vm32z</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 92 /vsib VGATHERDPD xmm1 {k1}, vm32x</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 92 /vsib VGATHERDPD ymm1 {k1}, vm32y</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 92 /vsib VGATHERDPD zmm1 {k1}, vm32y</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

A set of single-precision/double-precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element. The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

VGATHERDPS (EVEX encoded version)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    THEN DEST[i+31:i] ← MEM[BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
    k1[j] ← 0
    ELSE *DEST[i+31:i] remains unchanged*
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAX_VL-1:VL] ← 0

VGATHERDPD (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    THEN DEST[i+63:i] ← MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
    k1[j] ← 0
    ELSE *DEST[i+63:i] remains unchanged*
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VGATHERDPD __m512d _mm512_i32gather_pd(__m256i vdx, void * base, int scale);
VGATHERDPD __m512d _mm512_mask_i32gather_pd(__m512 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d _mm256_mmask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d _mm_mmask_i32gather_pd(__m128 d s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E12.
VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 C6 /vsib VGATHERPF0DPS vm32z (k1)</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C7 /vsib VGATHERPF0QPS vm64z (k1)</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C6 /vsib VGATHERPF0DPD vm32y (k1)</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C7 /vsib VGATHERPF0QPD vm64z (k1)</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into a location in the cache hierarchy specified by a locality hint (T0):

- T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits.
6. A #UD is signaled if the memory operand is encoded without the SIB byte.
Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by ‘mem’ into the cache level specified by ‘Level’; a request for exclusive/ownership is done if ‘State’ is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF0DPS (EVEX encoded version)
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

VGATHERPF0DPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

VGATHERPF0QPS (EVEX encoded version)
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR

VGATHERPF0QPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
**Intel C/C++ Compiler Intrinsic Equivalent**

VGATHERPF0DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF0QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12NP.
VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 C6 /vsib VGATHERPF1DPS vm32z {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C7 /vsib VGATHERPF1QPS vm64z {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C6 /vsib VGATHERPF1DPD vm32y {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C7 /vsib VGATHERPF1QPD vm64z {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into a location in the cache hierarchy specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.
Operation
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF1DPS (EVEX encoded version)
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR

VGATHERPF1DPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR

VGATHERPF1QPS (EVEX encoded version)
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR

VGATHERPF1QPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 64
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 0)
  FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF1DPD void_mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1DPS void_mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF1QPD void_mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1QPS void_mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12NP.
### VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64y</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 93 /vsib VGATHERQPD xmm1 {k1}, vm64x</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>T1S</td>
</tr>
</tbody>
</table>

### Description

A set of 8 single-precision/double-precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

This instruction does not perform AC checks, and so will never deliver an AC fault.

Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1, 2 or 4 byte displacement

**VGATHERQPS (EVEX encoded version)**

(KL, VL) = (2, 64), (4, 128), (8, 256)

FOR j ← 0 TO KL-1

    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
        k1[j] ← 0
        ELSE *DEST[i+31:i] remains unchanged*
        FI;
    ENDFOR

k1[MAX_KL-1:KL] ← 0
DEST[MAX_VL-1:VL/2] ← 0

**VGATHERQPD (EVEX encoded version)**

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

FOR j ← 0 TO KL-1

    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
        k1[j] ← 0
        ELSE *DEST[i+63:i] remains unchanged*
        FI;
    ENDFOR

k1[MAX_KL-1:KL] ← 0
DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VGATHERQPD __m512d _mm512_i64gather_pd( __m512i vdx, void * base, int scale);
VGATHERQPD __m512d _mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d _mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d _mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_i64gather_ps( __m512i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 _mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 _mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.
VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W0 90 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W0 91 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 90 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather dword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W0 91 /r</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather dword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero’ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.
VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

• If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
• The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the MSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
• Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• This instruction will cause a #UD if the address size attribute is 16-bit.
• This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
• This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminable number of times.
• The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VPGATHERDD (VEX.128 version)

FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i+31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i+31:i] ← 0;
  FI;
ENDFOR

MASK[VLMAX-1:128] ← 0;

FOR j ← 0 to 3
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i+31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i+31:i] ← 0;
ENDFOR

DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
VPGATHERQD (VEX.128 version)

\[
\text{FOR } j \leftarrow 0 \text{ to } 3 \\
\quad i \leftarrow j \times 32; \\
\quad \text{IF MASK}[31+i] \text{ THEN} \\
\quad \quad \text{MASK}[i+31:i] \leftarrow \text{FFFFFFFF}; \quad \text{// extend from most significant bit} \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \text{MASK}[i+31:i] \leftarrow 0; \\
\quad \quad \text{FI;}
\]
\[
\text{ENDFOR}
\]
\[
\text{MASK}[\text{VLMAX}-1:128] \leftarrow 0;
\]
\[
\text{FOR } j \leftarrow 0 \text{ to } 1 \\
\quad k \leftarrow j \times 64; \\
\quad i \leftarrow j \times 32; \\
\quad \text{DATA_ADDR} \leftarrow \text{BASE_ADDR} + (\text{SignExtend(VINDEX1}[k+63:k]) \times \text{SCALE} + \text{DISP}; \\
\quad \text{IF MASK}[31+i] \text{ THEN} \\
\quad \quad \text{DEST}[i+31:i] \leftarrow \text{FETCH_32Bits(DATA_ADDR);} \quad \text{// a fault exits the instruction} \\
\quad \quad \text{FI;}
\quad \quad \text{MASK}[i+31:i] \leftarrow 0;
\quad \text{ENDFOR}
\]
\[
\text{MASK}[127:64] \leftarrow 0; \\
\text{DEST}[\text{VLMAX}-1:64] \leftarrow 0;
\]
\[
\text{(non-masked elements of the mask register have the content of respective element cleared)}
\]

VPGATHERDD (VEX.256 version)

\[
\text{FOR } j \leftarrow 0 \text{ to } 7 \\
\quad i \leftarrow j \times 32; \\
\quad \text{IF MASK}[31+i] \text{ THEN} \\
\quad \quad \text{MASK}[i+31:i] \leftarrow \text{FFFFFFFF}; \quad \text{// extend from most significant bit} \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \text{MASK}[i+31:i] \leftarrow 0; \\
\quad \quad \text{FI;}
\quad \text{ENDFOR}
\]
\[
\text{FOR } j \leftarrow 0 \text{ to } 7 \\
\quad i \leftarrow j \times 32; \\
\quad \text{DATA_ADDR} \leftarrow \text{BASE_ADDR} + (\text{SignExtend(VINDEX1}[i+31:i]) \times \text{SCALE} + \text{DISP}; \\
\quad \text{IF MASK}[31+i] \text{ THEN} \\
\quad \quad \text{DEST}[i+31:i] \leftarrow \text{FETCH_32Bits(DATA_ADDR);} \quad \text{// a fault exits the instruction} \\
\quad \quad \text{FI;}
\quad \quad \text{MASK}[i+31:i] \leftarrow 0;
\quad \text{ENDFOR}
\]
\[
\text{(non-masked elements of the mask register have the content of respective element cleared)}
\]
VPGATHERQD (VEX.256 version)
FOR j ← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j ← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent
VPGATHERD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERD: __m256i _mm256_i32gather_epi32 ( int const * base, __m256i index, const int scale);
VPGATHERD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);
VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);
VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);
VPGATHERQD: __m256i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);
VPGATHERQD: __m256i _mm256_mask_i64gather_epi32 (__m256i src, int const * base, __m256i index, __m128i mask, const int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 12.
VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword with Signed Dword Indices

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 90 /vsib</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 90 /vsib</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 90 /vsib</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 90 /vsib</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 90 /vsib</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 90 /vsib</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM reg (w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into vector zmm1. The elements are specified via the VSIB (i.e., the index register is a zmm, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register (zmm1) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.
Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VININDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

**BASE_ADDR** stands for the memory operand base address (a GPR); may not exist

**VININDEX** stands for the memory operand vector of indices (a ZMM register)

**SCALE** stands for the memory operand scalar (1, 2, 4 or 8)

**DISP** is the optional 1, 2 or 4 byte displacement

**VPGATHERDD (EVEX encoded version)**

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

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\[i \leftarrow j * 32\]

\[\text{IF } k1[j]\]

\[\text{THEN } \text{DEST}[i+31:i] \leftarrow \text{MEM}[\text{BASE_ADDR} + \text{SignExtend(VINDEX}[i+31:i]) * \text{SCALE} + \text{DISP}], 1)\]

\[k1[j] \leftarrow 0\]

\[\text{ELSE *DEST}[i+31:i] \leftarrow \text{remains unchanged}^{*}; \text{Only merging masking is allowed}\]

\[\text{FI;}\]

ENDFOR

\[k1[\text{MAX_KL-1:KL}] \leftarrow 0\]

\[\text{DEST}[\text{MAX_VL-1:VL}] \leftarrow 0\]

**VPGATHERDQ (EVEX encoded version)**

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

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\[i \leftarrow j * 64\]

\[k \leftarrow j * 32\]

\[\text{IF } k1[j]\]

\[\text{THEN } \text{DEST}[i+63:i] \leftarrow \text{MEM}[\text{BASE_ADDR} + \text{SignExtend(VINDEX}[k+31:k]) * \text{SCALE} + \text{DISP}]\]

\[k1[j] \leftarrow 0\]

\[\text{ELSE *DEST}[i+63:i] \leftarrow \text{remains unchanged}^{*}; \text{Only merging masking is allowed}\]

\[\text{FI;}\]

ENDFOR

\[k1[\text{MAX_KL-1:KL}] \leftarrow 0\]

\[\text{DEST}[\text{MAX_VL-1:VL}] \leftarrow 0\]
**Intel C/C++ Compiler Intrinsic Equivalent**

VPGATHERDD __m512i _mm512_i32gather_epi32( __m512i vdx, void * base, int scale);
VPGATHERDD __m512i _mm512_mask_i32gather_epi32( __m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i _mm256_mmask_i32gather_epi32( __m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i _mm_mmask_i32gather_epi32( __m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_i32logather_epi64( __m256i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_mask_i32logather_epi64( __m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERDQ __m256i _mm256_mmask_i32logather_epi64( __m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i _mm_mmask_i32logather_epi64( __m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.
# VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices

## Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.DDS.128.66.0F38.W1 90 /r VPGATHERDQ xmm1, vm32x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.128.66.0F38.W1 91 /r VPGATHERQQ xmm1, vm64x, xmm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 90 /r VPGATHERDQ ymm1, vm32y, ymm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.DDS.256.66.0F38.W1 91 /r VPGATHERQQ ymm1, vm64y, ymm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg(R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

## Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.
VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

• If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
• The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
• Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• This instruction will cause a #UD if the address size attribute is 16-bit.
• This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
• This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
• The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.
Operation

DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VPGATHERDQ (VEX.128 version)
FOR j ← 0 to 1
  i ← j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  k ← j * 32;
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +63:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQQ (VEX.128 version)
FOR j ← 0 to 1
  i ← j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +63:i] ← 0;
ENDFOR
MASK[VLMAX-1:128] ← 0;
DEST[VLMAX-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element cleared)
VPGATHERQQ (VEX.256 version)

FOR j ← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i+63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i+63:i] ← 0;
    FI;
ENDFOR

FOR j ← 0 to 3
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i+63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i+63:i] ← 0;
ENDFOR

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERDQ (VEX.256 version)

FOR j ← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i+63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i+63:i] ← 0;
    FI;
ENDFOR

FOR j ← 0 to 3
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i+63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i+63:i] ← 0;
ENDFOR

(non-masked elements of the mask register have the content of respective element cleared)
Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDQ: __m128i __m128i_mm_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);
VPGATHERDQ: __m128i __m128i_mm_mask_i32gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);
VPGATHERDQ: __m256i __m256i_mm256_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);
VPGATHERDQ: __m256i __m256i_mm256_mask_i32gather_epi64 (__m256i src, __int64 const * base, __m128i index, __m256i mask, const int scale);
VPGATHERQQ: __m128i __m128i_mm_i64gather_epi64 (__int64 const * base, __m128i index, const int scale);
VPGATHERQQ: __m128i __m128i_mm_mask_i64gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);
VPGATHERQQ: __m256i __m256i_mm256_i64gather_epi64 (__int64 const * base, __m256i index, const int scale);
VPGATHERQQ: __m256i __m256i_mm256_mask_i64gather_epi64 (__m256i src, __int64 const * base, __m256i index, __m256i mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12.
VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

Description

A set of 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• Not valid with 16-bit effective addresses. Will deliver a #UD fault.
• These instructions do not accept zeroing-masking since the 0 values in \( k1 \) are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1). The instruction will #UD fault if the destination vector \( zmm1 \) is the same as index vector \( \text{VINDEX} \). The instruction will #UD fault if the \( k0 \) mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

\( \text{BASE ADDR} \) stands for the memory operand base address (a GPR); may not exist
\( \text{VINDEX} \) stands for the memory operand vector of indices (a ZMM register)
\( \text{SCALE} \) stands for the memory operand scalar (1, 2, 4 or 8)
\( \text{DISP} \) is the optional 1, 2 or 4 byte displacement

**VPGATHERQD (EVEX encoded version)**

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

FOR \( j \leftarrow 0 \) TO \( KL-1 \)
\( i \leftarrow j \ast 32 \)
\( k \leftarrow j \ast 64 \)
\( \text{IF } k1[j] \)
\( \text{THEN } \text{DEST}[i+31:i] \leftarrow \text{MEM}[\text{BASE_ADDR} + (\text{VINDEX}[k+63:k]) \ast \text{SCALE} + \text{DISP}], 1) \)
\( k1[j] \leftarrow 0 \)
\( \text{ELSE *DEST}[i+31:i] \leftarrow \text{remains unchanged* } ; \text{Only merging masking is allowed} \)
\( \text{FI; ENDFOR} \)
\( k1[\text{MAX}_KL-1:KL] \leftarrow 0 \)
\( \text{DEST}[\text{MAX}_VL-1:VL/2] \leftarrow 0 \)

**VPGATHERQQ (EVEX encoded version)**

\((KL, VL) = (2, 64), (4, 128), (8, 256)\)

FOR \( j \leftarrow 0 \) TO \( KL-1 \)
\( i \leftarrow j \ast 64 \)
\( \text{IF } k1[j] \)
\( \text{THEN } \text{DEST}[i+63:i] \leftarrow \text{MEM}[\text{BASE_ADDR} + (\text{VINDEX}[i+63:i]) \ast \text{SCALE} + \text{DISP}) \)
\( k1[j] \leftarrow 0 \)
\( \text{ELSE *DEST}[i+63:i] \leftarrow \text{remains unchanged* } ; \text{Only merging masking is allowed} \)
\( \text{FI; ENDFOR} \)
\( k1[\text{MAX}_KL-1:KL] \leftarrow 0 \)
\( \text{DEST}[\text{MAX}_VL-1:VL] \leftarrow 0 \)
**Intel C/C++ Compiler Intrinsic Equivalent**

VPGATHERQD __m256i _mm512_i64gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERQD __m256i _mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i _mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i _mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_i64gather_epi64( __m512i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQQ __m256i _mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQQ __m128i _mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.
VGETEXPPD—Convert Exponents of Packed DP FP Values to DP FP Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bitMode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 42 /r</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>AVX512F</td>
<td>Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 42 /r</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>AVX512F</td>
<td>Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 42 /r</td>
<td>V/V</td>
<td>AVX512F</td>
<td></td>
<td>Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Extracts the biased exponents from the normalized DP FP representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double-precision FP value and written to the corresponding qword elements of the destination operand (the first operand) as DP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

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

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-7.

The formula is:

\[
\text{GETEXP}(x) = \text{floor}(\log_2(|x|))
\]

Notation \(\text{floor}(x)\) stands for the greatest integer not exceeding real number \(x\).

Table 5-7. VGETEXPPD/SD Special Cases

<table>
<thead>
<tr>
<th>Input Operand</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = NaN</td>
<td>NaN(src1)</td>
<td>No Exceptions</td>
</tr>
<tr>
<td>0 &lt;</td>
<td>src1</td>
<td>&lt; INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= +INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= 0</td>
</tr>
</tbody>
</table>
**Operation**

NormalizeExpTinyDPFP(SRC[63:0])

```
{  
  // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
  Src.Jbit <- 0;
  Dst.exp <- 1;
  Dst.fraction <- SRC[51:0];
  WHILE(Src.Jbit = 0)  
  {  
    Src.Jbit <- Dst.fraction[51]; // Get the fraction MSB
    Dst.fraction <- Dst.fraction << 1; // One bit shift left
    Dst.exp--; // Decrement the exponent
  }
  Dst.fraction <- 0; // zero out fraction bits
  Dst.sign <- 1; // Return negative sign
  TMP[63:0] <- MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
  Return (TMP[63:0]);
}
```

ConvertExpDPFP(SRC[63:0])

```
{  
  Src.sign <- 0; // Zero out sign bit
  Src.exp <- SRC[62:52];
  Src.fraction <- SRC[51:0];
  // Check for NaN
  IF (SRC = NaN)
  {  
    IF ( SRC = SNAN ) SET IE;
    Return QNAN(SRC);
  }
  // Check for +INF
  IF (SRC = +INF) Return (SRC);
  // check if zero operand
  IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
  ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
  {  
    IF ((Src.exp = 0) AND (Src.fraction != 0))
    {  
      TMP[63:0] <- NormalizeExpTinyDPFP(SRC[63:0]); // Get Normalized Exponent
      Set #DE
    }
    ELSE // exponent value is correct
    {  
      Dst.fraction <- 0; // zero out fraction bits
      TMP[63:0] <- (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction);
    }
    TMP <- SAR(TMP, 52); // Shift Arithmetic Right
    TMP <- TMP - 1023; // Subtract Bias
    Return CvtI2D(TMP); // Convert INT to Double-Precision FP number
  }
}
```
VGETEXPPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+63:i] ← ConvertExpDPFP(SRC[63:0])
                ELSE
                    DEST[i+63:i] ← ConvertExpDPFP(SRC[i+63:i])
                    FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] ← 0
                FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd( __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd( __mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd( __mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 42 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 42 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 42 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Extracts the biased exponents from the normalized SP FP representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision FP value and written to the corresponding dword elements of the destination operand (the first operand) as SP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-8.

The formula is:

\[
\text{GETEXP}(x) = \text{floor}(\log_2(|x|))
\]

Notation \(\text{floorn}(x)\) stands for maximal integer not exceeding real number \(x\).

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

Table 5-8. VGETEXPPS/SS Special Cases

<table>
<thead>
<tr>
<th>Input Operand</th>
<th>Result</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>src1 = NaN</td>
<td>QNaN(src1)</td>
<td>No Exceptions</td>
</tr>
<tr>
<td>0 &lt;</td>
<td>src1</td>
<td>&lt; INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= +INF</td>
</tr>
<tr>
<td></td>
<td>src1</td>
<td>= 0</td>
</tr>
</tbody>
</table>
Figure 5-14 illustrates the VGETEXPPS functionality on input values with normalized representation.

**Operation**

### NormalizeExpTinySPFP(SRC[31:0])

```c
// Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
Src.Jbit ↔ 0;
Dst.exp ↔ 1;
Dst.fraction ↔ SRC[22:0];
WHILE(Src.Jbit = 0)
{
    Src.Jbit ↔ Dst.fraction[22]; // Get the fraction MSB
    Dst.fraction ↔ Dst.fraction << 1; // One bit shift left
    Dst.exp--; // Decrement the exponent
}
Dst.fraction ↔ 0; // zero out fraction bits
Dst.sign ↔ 1; // Return negative sign
TMP[31:0] ↔ MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
Return (TMP[31:0]);
```

### ConvertExpSPFP(SRC[31:0])

```c
Src.sign ↔ 0; // Zero out sign bit
Src.exp ↔ SRC[30:23];
Src.fraction ↔ SRC[22:0];
// Check for NaN
IF (SRC = NaN)
{
    IF (SRC = SNAN) SET IE;
    Return QNAN(SRC);
}
// Check for +INF
IF (SRC = +INF) Return (SRC);

// check if zero operand
IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
{
```
IF ((Src.exp = 0) AND (Src.fraction /= 0))
{
    TMP[31:0] \leftarrow \text{NormalizeExpTinySPFP}(SRC[31:0]); \quad \text{// Get Normalized Exponent}
    Set #DE
}
ELSE \quad \text{// exponent value is correct}
{
    Dst.fraction \leftarrow 0; \quad \text{// zero out fraction bits}
    TMP[31:0] \leftarrow (Src.sign \ll 31) \text{ OR } (Src.exp \ll 23) \text{ OR } (Src.fraction);
}
    TMP \leftarrow \text{SAR}(TMP, 23); \quad \text{// Shift Arithmetic Right}
    TMP \leftarrow TMP - 127; \quad \text{// Subtract Bias}
    \text{Return CvtI2D}(TMP); \quad \text{// Convert INT to Single-Precision FP number}
}

\text{VGETEXPPS (EVEX encoded versions)}
(KL, VL) = (4, 128), (8, 256), (16, 512)
\text{FOR } j \leftarrow 0 \text{ TO } KL-1
    i \leftarrow j \times 32
    \text{IF } k1[j] \text{ OR *no writemask*}
        \text{THEN}
            \text{IF (EVEX.b = 1) AND (SRC *is memory*)}
            \text{THEN}
                \text{DEST}[i+31:i] \leftarrow \text{ConvertExpSPFP}(SRC[31:0])
            \text{ELSE}
                \text{DEST}[i+31:i] \leftarrow \text{ConvertExpSPFP}(SRC[i+31:i])
            \text{FI;}
        \text{ELSE}
            \text{IF *merging-masking*} ; \text{merging-masking}
            \text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged*}
            \text{ELSE} \quad ; \text{zeroing-masking}
                \text{DEST}[i+31:i] \leftarrow 0
            \text{FI}
        \text{FI;}
\text{ENDFOR}
\text{DEST}[\text{MAX}_\text{VL}-1:1] \leftarrow 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPS __m512 __m512_getexp_ps( __m512 a);
VGETEXPPS __m512 __mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a);
VGETEXPPS __m512 __mm512_maskz_getexp_ps( __mmask16 k, __m512 a);
VGETEXPPS __m512 __mm512_getexp_round_ps( __m512 a, int sae);
VGETEXPPS __m512 __mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae);
VGETEXPPS __m512 __mm512_maskz_getexp_round_ps( __mmask16 k, __m512 a, int sae);
VGETEXPPS __m256 __mm256_getexp_ps(__mm256 a);
VGETEXPPS __m256 __mm256_mask_getexp_ps(__mm256 s, __mmask8 k, __m256 a);
VGETEXPPS __m256 __mm256_maskz_getexp_ps( __mmask8 k, __m256 a);
VGETEXPPS __m128 __mm128_getexp_ps(__m128 a);
VGETEXPPS __m128 __mm128_mask_getexp_ps(__m128 s, __mmask8 k, __m128 a);
VGETEXPPS __m128 __mm128_maskz_getexp_ps( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W1 43 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the biased exponent (bits 62:52) of the low double-precision floating-point value in xmm3/m64 to a DP FP value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Extracts the biased exponent from the normalized DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be an XMM register, the source operand can be a XMM register or a float64 memory location. The low quadword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-7.

The formula is:

\[ \text{GETEXP}(x) = \text{floor}(\log_2(|x|)) \]

Notation \( \text{floor}(x) \) stands for maximal integer not exceeding real number \( x \).

### Operation

// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

**VGETEXPSD (EVEX encoded version)**

IF k1[0] OR *no writemask*

THEN DEST[63:0] ← ConvertExpDPFP(SRC2[63:0])

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged* ; zeroing-masking

ELSE

DEST[63:0] ← 0

FI

FI:

DEST[127:64] ← SRC1[127:64]

DEST[MAX_VL-1:128] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VGETEXPSD __m128d __m128d_mm_getexp_sd(__m128d a, __m128d b);
VGETEXPSD __m128d __m128d_mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d __m128d_mm_maskz_getexp_sd(__mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d __m128d_mm_getexp_round_sd(__m128d a, __m128d b, int sae);
VGETEXPSD __m128d __m128d_mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXPSD __m128d __m128d_mm_maskz_getexp_round_sd(__mmask8 k, __m128d a, __m128d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E3.
VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LG.66.0F38.W0 43 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a SP FP value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Extracts the biased exponent from the normalized SP FP representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision FP value and written to the destination operand (the first operand) as SP FP numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location. The the low doubleword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-8.

The formula is:

\[ \text{GETEXP}(x) = \text{floor}(\log_2(|x|)) \]

Notation \( \text{floor}(x) \) stands for maximal integer not exceeding real number \( x \).

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

Operation

// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

**VGETEXPSS (EVEX encoded version)**

IF k1[0] OR *no writemask*

\[ \text{THEN} \quad \text{DEST}[31:0] \leftarrow \text{ConvertExpDPFP(SRC2[31:0])} \]

ELSE

\[ \text{IF} *\text{merging-masking}*, \quad \text{DEST}[31:0] \text{ remains unchanged}^* \]

\[ \text{ELSE} \quad \text{DEST}[31:0] \leftarrow 0 \]

\[ \text{FI} \]

\[ \text{FI} \]

ENDFOR

\[ \text{DEST}[127:32] \leftarrow \text{SRC1}[127:32] \]

\[ \text{DEST}[\text{MAX_VL}-1:128] \leftarrow 0 \]
Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSS __m128 _mm_getexp_ss(__m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss(__mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss(__m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss(__mmask8 k, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.
VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 26 / r ib VGETMANTPD xmm1 {k1}[z], xmm2/m128/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 26 / r ib VGETMANTPD ymm1 {k1}[z], ymm2/m256/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 26 / r ib VGETMANTPD zmm1 {k1}[z], zmm2/m512/m64bcst{sae}, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Convert double-precision floating values in the source operand (the second operand) to DP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

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

**Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS**

For each input DP FP value x, The conversion operation is:

\[ \text{GetMant}(x) = \pm 2^k |x.\text{significand}| \]

where:

\[ 1 \leq |x.\text{significand}| < 2 \]

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.
If interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted DP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

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

### Table 5-9. GetMant() Special Float Values Behavior

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
<th>Exceptions / Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>QNaN(SRC)</td>
<td>Ignore interv</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If (SRC = SNaN) then #IE</td>
</tr>
<tr>
<td>+∞</td>
<td>1.0</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>+0</td>
<td>1.0</td>
<td>Ignore interv</td>
</tr>
<tr>
<td>-0</td>
<td>IF (SC[0]) THEN +1.0 ELSE -1.0</td>
<td>Ignore interv</td>
</tr>
</tbody>
</table>
| -∞      | IF (SC[1]) THEN {QNaN_Indefinite}
|         | ELSE {
|         | IF (SC[0]) THEN +1.0 ELSE -1.0  | Ignore interv                               |
|         |                                 | If (SC[1]) then #IE                        |
| negative| SC[1] ? QNaN_Indefinite : Getmant(SRC) | If (SC[1]) then #IE                        |
Operation

GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[63];  // Get sign bit
    Dst.exp ← SRC[62:52];  // Get original exponent value
    Dst.fraction ← SRC[51:0];  // Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 07FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 07FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {
        IF (SRC = SNaN) {Set #IE; Return QNAN(SRC);}
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand))
    {
        Dst.exp ← 03FFh;  // Override exponent with BIAS
        Return ((Dst.sign<<63) | (Dst.exp<<52) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[63] = 1) AND SignCtrl[1])
    {
        Set #IE;
        Return QNaN_Indefinite;
    }
    // Checking for denormal operands
    IF (DenormOperand)
    {
        IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
        ELSE
        {
            // Jbit is the hidden integral bit. Zero in case of denormal operand.
            Src.Jbit ← 0;  // Zero Src Jbit
            Dst.exp ← 03FFh;  // Override exponent with BIAS
            WHILE (Src.Jbit = 0) {
                Src.Jbit ← Dst.fraction[51];  // Get the fraction MSB
                Dst.fraction ← (Dst.fraction << 1);  // Start normalizing the mantissa
                Dst.exp--;  // Adjust the exponent
            }
            SET #DE;  // Set DE bit
        }
    }
    // At this point, Dst.exp has the correct result. Form the final destination
    Unbiased.exp ← Dst.exp - 03FFh;  // subtract the bias from exponent
    IsOddExp ← Unbiased.exp[0];  // recognized unbiased ODD exponent
    SignalingBit ← Dst.fraction[51];
    CASE (interv[1:0])
    00: Dst.exp ← 03FFh;  // This is the bias
    01: Dst.exp ← (IsOddExp ? 03FEh : 03FFh);  // either bias-1, or bias
    10: Dst.exp ← 03FEh;  // bias-1
    11: Dst.exp ← (SignalingBit ? 03FEh : 03FFh);  // either bias-1, or bias
    ESCA
    // At this point Dst.exp has the correct result. Form the final destination
    DEST[63:0] ← (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
    Return (DEST);
}
INSTRUCTION SET REFERENCE, V-Z

VGETMANTPD (EVEX encoded versions)

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

FOR \(j \leftarrow 0\) TO \(KL-1\)

\[i \leftarrow j \times 64\]

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

THEN

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

THEN

\[\text{DEST}[i\!+\!63:i] \leftarrow \text{GetNormalizedMantissaDP}(\text{SRC}[63:0], \text{sc}, \text{interv})\]

ELSE

\[\text{DEST}[i\!+\!63:i] \leftarrow \text{GetNormalizedMantissaDP}(\text{SRC}[i\!+\!63:i], \text{sc}, \text{interv})\]

FI;

ELSE

IF *merging-masking*  ; merging-masking

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

ELSE  ; zeroing-masking

\[\text{DEST}[i\!+\!63:i] \leftarrow 0\]

FI

ENDFOR

\[\text{DEST}[\text{MAX}_VL-1:VL] \leftarrow 0\]

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTPD __m512d _mm512_getmant_pd(__m512d a, enum intv, enum sgn);

VGETMANTPD __m512d _mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);

VGETMANTPD __m512d _mm512_maskz_getmant_pd(__mmask8 k, __m512d a, enum intv, enum sgn);

VGETMANTPD __m512d _mm512_getmant_round_pd(__m512d a, enum intv, enum sgn, int r);

VGETMANTPD __m512d _mm512_maskz_getmant_round_pd(__mmask8 k, __m512d a, enum intv, enum sgn, int r);

VGETMANTPD __m256d _mm256_getmant_pd(__m256d a, enum intv, enum sgn);

VGETMANTPD __m256d _mm256_mask_getmant_pd(__mmask8 k, __m256d a, enum intv, enum sgn);

VGETMANTPD __m256d _mm256_maskz_getmant_pd(__mmask8 k, __m256d a, enum intv, enum sgn);

VGETMANTPD __m256d _mm256_getmant_round_pd(__m256d a, enum intv, enum sgn, int r);

VGETMANTPD __m256d _mm256_maskz_getmant_round_pd(__mmask8 k, __m256d a, enum intv, enum sgn, int r);

VGETMANTPD __m128d _mm_getmant_pd(__m128d a, enum intv, enum sgn);

VGETMANTPD __m128d _mm_mask_getmant_pd(__mmask8 k, __m128d a, enum intv, enum sgn);

VGETMANTPD __m128d _mm_maskz_getmant_pd(__mmask8 k, __m128d a, enum intv, enum sgn);

VGETMANTPD __m128d _mm_maskz_getmant_round_pd(__mmask8 k, __m128d a, enum intv, enum sgn);

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E2.

#UD  \text{If EVEX.vvvv} \neq 1111B.
VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector

Description

Convert single-precision floating values in the source operand (the second operand) to SP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input SP FP value \( x \), the conversion operation is:

\[
\text{GetMant}(x) = \pm 2^k |x.\text{significand}|
\]

where:

\[ 1 \leq |x.\text{significand}| < 2 \]

Unbiased exponent \( k \) depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv ≠ 0 then \( k = -1 \), otherwise \( K = 0 \). The encoded value of \( \text{imm8}[1:0] \) and the sign control are shown in Figure 5-15.

Each converted SP FP result is encoded according to the sign control, the unbiased exponent \( k \) (adding bias) and a mantissa normalized to the range specified by interv.

The \( \text{GetMant()} \) function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 26 /r</td>
<td>ib VGETMANTPS xmm1 k1]{z}, xmm2/m128/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 26 /r</td>
<td>ib VGETMANTPS ymm1 k1]{z}, ymm2/m256/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 26 /r</td>
<td>ib VGETMANTPS zmm1 k1]{z}, zmm2/m512/m32bcst{sae}, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
</tr>
</tbody>
</table>
Operation
GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[31]; // Get sign bit
    Dst.exp ← SRC[30:23]; // Get original exponent value
    Dst.fraction ← SRC[22:0]; // Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 0FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 0FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {
        IF (SRC = SNaN) {Set #IE;}
        Return QNAN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand)
    {
        Dst.exp ← 07Fh; // Override exponent with BIAS
        Return ((Dst.sign<<31) | (Dst.exp<<23) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[31] = 1) AND SignCtrl[1])
    {
        Set #IE;
        Return QNaN_Indefinite;
    }
    // Checking for denormal operands
    IF (DenormOperand)
    {
        IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
        ELSE
        {
            Jbit is the hidden integral bit. Zero in case of denormal operand.
            Src.Jbit ← 0; // Zero Src Jbit
            Dst.exp ← 07Fh; // Override exponent with BIAS
            WHILE (Src.Jbit = 0) {
                // normalize mantissa
                Src.Jbit ← Dst.fraction[22]; // Get the fraction MSB
                Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
                Dst.exp--; // Adjust the exponent
            }
            SET #DE; // Set DE bit
        }
    }
    // At this point, Dst.fraction is normalized.
    // Checking for exponent response
    Unbiased.exp ← Dst.exp - 07Fh; // subtract the bias from exponent
    IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
    SignalingBit ← Dst.fraction[22]; // recognized signaling bit
    CASE (Interv[1:0])
    00: Dst.exp ← 07Fh; // This is the bias
    01: Dst.exp ← (IsOddExp) ? 07Eh : 07Fh; // either bias-1, or bias
    10: Dst.exp ← 07Eh; // bias-1
    11: Dst.exp ← (SignalingBit) ? 07Eh : 07Fh; // either bias-1, or bias
    ESAC
    // Form the final destination
    DEST[31:0] ← (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
}
Return (DEST);
}

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];

**VGETMANTPS (E VEX encoded versions)**

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

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN
        DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[31:0], sc, interv)
      ELSE
        DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[i+31:i], sc, interv)
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0

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

VGETMANTPS __m512 _mm512_getmant_ps( __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_mask_getmant_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_maskz_getmant_ps( __mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_getmant_round_ps( __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_mask_getmant_round_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_maskz_getmant_round_ps( __mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m256 _mm256_getmant_ps( __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_mask_getmant_ps(__m256 s, __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_maskz_getmant_ps( __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_getmant_ps( __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_mask_getmant_ps(__m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_maskz_getmant_ps( __mmask8 k, __m128 a, enum intv, enum sgn);

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.
VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W1 27 / r ib VGETMANTSD xmm1 [k1];z, xmm2, xmm3/m64{sa}e, imm8</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract the normalized mantissa of the low float64 element in xmm3/m64 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Convert the double-precision floating values in the low quadword element of the second source operand (the third operand) to DP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

\[
\text{GetMant}(x) = \pm 2^k |x.\text{significand}|
\]

where:

\[1 \leq |x.\text{significand}| < 2\]

Unbiased exponent \(k\) depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

If interv ! = 0 then \(k = -1\), otherwise \(K = 0\). The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted DP FP result is encoded according to the sign control, the unbiased exponent \(k\) (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into xmm1. Elements in xmm1 with the corresponding bit clear in k1 retain their previous values.
Operation

// GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];

VGETMANTSD (EVEX encoded version)
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ←
        GetNormalizedMantissaDP(SRC2[63:0], sc, interv)
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged* 
    ELSE ; zeroing-masking
        DEST[63:0] ← 0
    FI
FI
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VGETMANTSD __m128d_mm_getmant_sd( __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d_mm_mask_getmant_sd( __m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d_mm_maskz_getmant_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d_mm_getmant_round_sd( __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d_mm_mask_getmant_round_sd( __m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d_mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);

SIMD Floating-Point Exceptions
Denormal, Invalid

Other Exceptions
See Exceptions Type E3.
VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 (k1)[z], xmm2, xmm3/m32{sa}, imm8</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to SP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$\text{GetMant}(x) = \pm2^k|x.\text{significand}|$$

where:

$$1 <= |x.\text{significand}| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.
**Operation**

// GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];

**VGETMANTSS (EVEX encoded version)**

IF k1[0] OR *no writemask*

THEN DEST[31:0] ←

GetNormalizedMantissaSP(SRC2[31:0], sc, interv)

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged* 

ELSE ; zeroing-masking

DEST[31:0] ← 0
FI

Ft:
DEST[127:32] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

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

VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E3.
## VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF32X4 ymm1 [k1][z], ymm2, xmm3/m128, imm8</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 18 /r ib VINSERTF32X4 zmm1 [k1][z], zmm2, xmm3/m128, imm8</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF64X2 ymm1 [k1][z], ymm2, xmm3/m128, imm8</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 18 /r ib VINSERTF64X2 zmm1 [k1][z], zmm2, xmm3/m128, imm8</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 1A /r ib VINSERTF32X8 zmm1 [k1][z], zmm2, ymm3/m256, imm8</td>
<td>T8</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 1A /r ib VINSERTF64X4 zmm1 [k1][z], zmm2, ymm3/m256, imm8</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 256 bits of packed double-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>T2, T4, T8</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

VINSERTF128/VINSERTF32x4 and VINSERTF64x2 insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF64x2: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF32x8 and VINSERTF64x4 inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.
**Operation**

**VINSERTF32x4 (EVEX encoded versions)**

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

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TEMP_DEST[127:0] ← SRC2[127:0]
1: TEMP_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TEMP_DEST[127:0] ← SRC2[127:0]
01: TEMP_DEST[255:128] ← SRC2[127:0]
10: TEMP_DEST[383:256] ← SRC2[127:0]
11: TEMP_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j * 32

IF k1[j] OR *no writemask*

THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAX_VL-1:VL] ← 0

**VINSERTF64x2 (EVEX encoded versions)**

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

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TEMP_DEST[127:0] ← SRC2[127:0]
1: TEMP_DEST[255:128] ← SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TEMP_DEST[127:0] ← SRC2[127:0]
01: TEMP_DEST[255:128] ← SRC2[127:0]
10: TEMP_DEST[383:256] ← SRC2[127:0]
11: TEMP_DEST[511:384] ← SRC2[127:0]

ESAC.

FI;

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]

ELSE
IF *merging-masking* ; merging-masking 
THEN *DEST[i+63:i] remains unchanged* 
ELSE ; zeroing-masking 
DEST[i+63:i] \leftarrow 0 
FI 
FI; 
ENDFOR 
DEST[MAX_VL-1:VL] \leftarrow 0 

**VINSERTF32x8 (EVEX.U1.512 encoded version)** 

\[
\text{TEMP_DEST}[VL-1:0] \leftarrow \text{SRC1}[VL-1:0] \\
\text{CASE (imm8[0]) OF} \\
\quad 0: \text{TMP_DEST}[255:0] \leftarrow \text{SRC2}[255:0] \\
\quad 1: \text{TMP_DEST}[511:256] \leftarrow \text{SRC2}[255:0] \\
\text{ESAC.} \\
\]

\[
\text{FOR } j \leftarrow 0 \text{ TO } 15 \\
\quad i \leftarrow j \times 32 \\
\text{IF } k1[j] \text{ OR *no writemask*} \\
\quad \text{THEN } \text{DEST}[i+31:i] \leftarrow \text{TMP_DEST}[i+31:i] \\
\quad \text{ELSE} \\
\quad \quad \text{IF *merging-masking*} ; \text{merging-masking} \\
\quad \quad \quad \quad \text{THEN *DEST}[i+31:i] \text{ remains unchanged*} \\
\quad \quad \quad \quad \text{ELSE} ; \text{zeroing-masking} \\
\quad \quad \quad \quad \quad \quad \quad \text{DEST}[i+31:i] \leftarrow 0 \\
\quad \quad \quad \quad \FI \\
\quad \FI; \\
\ENDFOR \\
\text{DEST[MAX_VL-1:VL] \leftarrow 0} 
\]

**VINSERTF64x4 (EVEX.512 encoded version)**

VL = 512 
\[
\text{TEMP_DEST}[VL-1:0] \leftarrow \text{SRC1}[VL-1:0] \\
\text{CASE (imm8[0]) OF} \\
\quad 0: \text{TMP_DEST}[255:0] \leftarrow \text{SRC2}[255:0] \\
\quad 1: \text{TMP_DEST}[511:256] \leftarrow \text{SRC2}[255:0] \\
\text{ESAC.} \\
\]

\[
\text{FOR } j \leftarrow 0 \text{ TO } 7 \\
\quad i \leftarrow j \times 64 \\
\text{IF } k1[j] \text{ OR *no writemask*} \\
\quad \text{THEN } \text{DEST}[i+63:i] \leftarrow \text{TMP_DEST}[i+63:i] \\
\quad \text{ELSE} \\
\quad \quad \text{IF *merging-masking*} ; \text{merging-masking} \\
\quad \quad \quad \quad \text{THEN *DEST}[i+63:i] \text{ remains unchanged*} \\
\quad \quad \quad \quad \text{ELSE} ; \text{zeroing-masking} \\
\quad \quad \quad \quad \quad \quad \quad \text{DEST}[i+63:i] \leftarrow 0 \\
\quad \quad \quad \quad \FI \\
\quad \FI; \\
\ENDFOR \\
\text{DEST[MAX_VL-1:VL] \leftarrow 0} 
\]
**VINSERTF128 (VEX encoded version)**

TEMP[255:0] ← SRC1[255:0]

CASE (imm8[0]) OF
  0: TEMP[127:0] ← SRC2[127:0]
  1: TEMP[255:128] ← SRC2[127:0]
ESAC

DEST ← TEMP

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

VINSERTF32x4 __m512 _mm512_insertf32x4( __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 __mm512_mask_insertf32x4( __m512 s, __mmask16 k, __m512 a, __m128 b, int imm);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Exceptions Type 6; additionally
#UD  If VEX.L = 0.
EVEX-encoded instruction, see Exceptions Type E6NF.
**Instruction Set Reference, V-Z**

**INSTRUCTION SET REFERENCE, V-Z**

**VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.0F3A.W0 3B / ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 3B / ib</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 3B / ib</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 3B / ib</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 3B / ib</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 3A / ib</td>
<td>T8</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 3A / ib</td>
<td>T4</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>T2, T4, T8</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7 bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.
### Operation

**VINSERTI32x4 (EVEX encoded versions)**

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

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256
   CASE (imm8[0]) OF
      0: TMP_DEST[127:0] ← SRC2[127:0]
      1: TMP_DEST[255:128] ← SRC2[127:0]
   ESAC.

FI;

IF VL = 512
   CASE (imm8[1:0]) OF
      00: TMP_DEST[127:0] ← SRC2[127:0]
      01: TMP_DEST[255:128] ← SRC2[127:0]
   ESAC.

FI;

FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
   ELSE
      IF *merging-masking*
         THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+31:i] ← 0
      FI
   FI

ENDFOR

DEST[MAX_VL-1:VL] ← 0

**VINSERTI64x2 (EVEX encoded versions)**

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

TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]

IF VL = 256
   CASE (imm8[0]) OF
      0: TMP_DEST[127:0] ← SRC2[127:0]
   ESAC.

FI;

IF VL = 512
   CASE (imm8[1:0]) OF
      00: TMP_DEST[127:0] ← SRC2[127:0]
      01: TMP_DEST[255:128] ← SRC2[127:0]
   ESAC.

FI;

FOR j ← 0 TO KL-1
   i ← j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
   ELSE
IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
    DEST[i+63:i] \leftarrow 0
FI

ENDFOR
DEST[MAX_VL-1:VL] \leftarrow 0

VINSERTI32x8 (EVEX.U1.512 encoded version)
TEMP_DEST[VL-1:0] \leftarrow SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] \leftarrow SRC2[255:0]
  1: TMP_DEST[511:256] \leftarrow SRC2[255:0]
ESAC.

FOR j \leftarrow 0 TO 15
    i \leftarrow j * 32
    IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] \leftarrow TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] \leftarrow 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] \leftarrow 0

VINSERTI64x4 (EVEX.512 encoded version)
VL = 512
TEMP_DEST[VL-1:0] \leftarrow SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] \leftarrow SRC2[255:0]
  1: TMP_DEST[511:256] \leftarrow SRC2[255:0]
ESAC.

FOR j \leftarrow 0 TO 7
    i \leftarrow j * 64
    IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] \leftarrow TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] \leftarrow 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] \leftarrow 0
VINSERTI128
TEMP[255:0] ← SRC1[255:0]
CASE (imm8[0]) OF
  0: TEMP[127:0] ← SRC2[127:0]
  1: TEMP[255:128] ← SRC2[127:0]
ESAC
DEST ← TEMP

Intel C/C++ Compiler Intrinsic Equivalent
VINSERTI32x4 _mm512i _inserti32x4( __m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i _mask_inserti32x4( __m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i _maskz_inserti32x4( __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 _m256i _mm256_inserti32x4( __m256i a, __m128i b, int imm);
VINSERTI32x4 _m256i _mm256_mask_inserti32x4( __m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x4 _m256i _mm256_maskz_inserti32x4( __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x8 _m512i _mm512_inserti32x8( __m512i a, __m128i b, int imm);
VINSERTI32x8 _m512i _mm512_mask_inserti32x8( __m512i s, __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI32x8 _m512i _mm512_maskz_inserti32x8( __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI64x2 _m512i _mm512_inserti64x2( __m512i a, __m128i b, int imm);
VINSERTI64x2 _m512i _mm512_mask_inserti64x2( __m512i s, __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 _m512i _mm512_maskz_inserti64x2( __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x4 _m512i _mm512_inserti64x4( __m512i a, __m128i b, int imm);
VINSERTI64x4 _m512i _mm512_mask_inserti64x4( __m512i s, __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI64x4 _m512i _mm512_maskz_inserti64x4( __mmask8 m, __m512i a, __m256i b, int imm);
VINSERTI128 _m256i _mm256_insertf128_si256( __m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instruction, see Exceptions Type 6; additionally
#UD If VEX.L = 0.
EVEX-encoded instruction, see Exceptions Type E6NF.
VMASKMOV—Conditional SIMD Packed Loads and Stores

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 2C /r VMASKMOVPS xmm1, xmm2, m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally load packed single-precision values from m128 using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 2C /r VMASKMOVPS ymm1, ymm2, m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally load packed single-precision values from m256 using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 2D /r VMASKMOVPD xmm1, xmm2, m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally load packed double-precision values from m128 using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 2D /r VMASKMOVPD ymm1, ymm2, m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally load packed double-precision values from m256 using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 2E /r VMASKMOVPS m128, xmm1, xmm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally store packed single-precision values from xmm2 using mask in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 2E /r VMASKMOVPS m256, ymm1, ymm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally store packed single-precision values from ymm2 using mask in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 2F /r VMASKMOVPD m128, xmm1, xmm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally store packed double-precision values from xmm2 using mask in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 2F /r VMASKMOVPD m256, ymm1, ymm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX</td>
<td>Conditionally store packed double-precision values from ymm2 using mask in ymm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>ModRM:reg (w)</th>
<th>VEX.vvvv (r)</th>
<th>ModRM:r/m (r)</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVR</td>
<td>ModRM:r/m (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.
In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

**Operation**

**VMASKMOVPS - 128-bit load**

DEST[31:0] <- IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] <- IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] <- IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] <- IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[VLMAX-1:128] <- 0

**VMASKMOVPS - 256-bit load**

DEST[31:0] <- IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] <- IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] <- IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] <- IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] <- IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] <- IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] <- IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] <- IF (SRC1[255]) Load_32(mem + 28) ELSE 0

**VMASKMOVPS - 128-bit store**

IF (SRC1[31]) DEST[31:0] <- SRC2[31:0]
IF (SRC1[63]) DEST[63:32] <- SRC2[63:32]
IF (SRC1[95]) DEST[95:64] <- SRC2[95:64]
IF (SRC1[127]) DEST[127:96] <- SRC2[127:96]

**VMASKMOVPS - 256-bit store**

IF (SRC1[31]) DEST[31:0] <- SRC2[31:0]
IF (SRC1[63]) DEST[63:32] <- SRC2[63:32]
IF (SRC1[95]) DEST[95:64] <- SRC2[95:64]
IF (SRC1[127]) DEST[127:96] <- SRC2[127:96]
IF (SRC1[159]) DEST[159:128] <- SRC2[159:128]
IF (SRC1[255]) DEST[255:224] <- SRC2[255:224]
VMASKMOV—Conditional SIMD Packed Loads and Stores

VMASKMOVPD - 128-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]

VMASKMOVVPD - 256-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

__m256 _mm256_maskload_ps(float const *a, __m256i mask)
void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)
__m256d _mm256_maskload_pd(double *a, __m256i mask);
void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);
__m128 _mm128_maskload_ps(float const *a, __m128i mask)
void _mm128_maskstore_ps(float *a, __m128i mask, __m128 b)
__m128d _mm128_maskload_pd(double *a, __m128i mask);
void _mm128_maskstore_pd(double *a, __m128i mask, __m128d b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 6 (No AC# reported for any mask bit combinations);
additionally
#UD If VEX.W = 1.
VPBLENDD — Blend Packed Dwords

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F3A.W0 02 /r ib VPBLENDD xmm1, xmm2, xmm3/m128, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Select dwords from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F3A.W0 02 /r ib VPBLENDD ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Select dwords from ymm2 and ymm3/m256 from mask specified in imm8 and store the values into ymm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Dword elements from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

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

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

**Operation**

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

IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0] 
ELSE DEST[31:0] ← SRC1[31:0] 
ELSE DEST[95:64] ← SRC1[95:64] 
ELSE DEST[127:96] ← SRC1[127:96] 
ELSE DEST[159:128] ← SRC1[159:128] 
ELSE DEST[255:224] ← SRC1[255:224]
VPBLENDD (VEX.128 encoded version)

IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
ELSE DEST[95:64] ← SRC1[95:64]
ELSE DEST[127:96] ← SRC1[127:96]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDD: __m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)
VPBLENDD: __m256i _mm256_blend_epi32 (__m256i v1, __m256i v2, const int mask)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD If VEX.W = 1.
VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 66 /r VPBLENDMB xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 66 /r VPBLENDMB ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 66 /r VPBLENDMB zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 66 /r VPBLENDMW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 66 /r VPBLENDMW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 66 /r VPBLENDMW zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).
Operation

**VPBLENDMB (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1
   i ← j * 8
   IF k1[j] OR *no writemask*
      THEN DEST[i+7:i] ← SRC2[i+7:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN DEST[i+7:i] ← SRC1[i+7:i]
      ELSE ; zeroing-masking
         DEST[i+7:i] ← 0
      FI;
   FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0;

**VPBLENDMW (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1
   i ← j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] ← SRC2[i+15:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN DEST[i+15:i] ← SRC1[i+15:i]
      ELSE ; zeroing-masking
         DEST[i+15:i] ← 0
      FI;
   FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

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

**VPBLENDMB**

- _mm512_mask_blend_epi8(__m64 m, __m512i a, __m512i b);
- _mm256_mask_blend_epi8(__m32 m, __m256i a, __m256i b);
- _mm_mask_blend_epi8(__m16 m, __m128i a, __m128i b);

**VPBLENDMW**

- _mm512_mask_blend_epi16(__m32 m, __m512i a, __m512i b);
- _mm256_mask_blend_epi16(__m16 m, __m256i a, __m256i b);
- _mm_mask_blend_epi16(__m8 m, __m128i a, __m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.
VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

**Description**

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModR:reg (w)</td>
<td>EVEX.vvvv</td>
<td>ModR:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 64 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 64 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 64 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend doubleword integer vector zmm2 and doubleword vector zmm3/m128/m32bcst and store the result in zmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 64 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 64 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 64 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend quadword integer vector zmm2 and quadword vector zmm3/m128/m64bcst and store the result in zmm1, under control mask.</td>
</tr>
</tbody>
</table>
Operation

VPBLENDMD (EVEX encoded versions)

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

FOR \(j \leftarrow 0 \) TO \( KL-1 \)
    \( i \leftarrow j \times 32 \)
    IF \( k1[j] \) OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST\([i+31:i]\) \leftarrow SRC2[31:0]
                ELSE
                    DEST\([i+31:i]\) \leftarrow SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN DEST\([i+31:i]\) \leftarrow SRC1\([i+31:i]\]
                ELSE ; zeroing-masking
                    DEST\([i+31:i]\) \leftarrow 0
            FI;
        FI;
    FI;
ENDFOR

DEST\([MAX\_VL-1:VL]\) \leftarrow 0;

VPBLENDMD (EVEX encoded versions)

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

FOR \(j \leftarrow 0 \) TO \( KL-1 \)
    \( i \leftarrow j \times 32 \)
    IF \( k1[j] \) OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST\([i+31:i]\) \leftarrow SRC2[31:0]
                ELSE
                    DEST\([i+31:i]\) \leftarrow SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN DEST\([i+31:i]\) \leftarrow SRC1\([i+31:i]\]
                ELSE ; zeroing-masking
                    DEST\([i+31:i]\) \leftarrow 0
            FI;
        FI;
ENDFOR

DEST\([MAX\_VL-1:VL]\) \leftarrow 0;
Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMD __m512i __mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);
VPBLENDMD __m256i __mm256_mask_blend_epi32(__mmask8 m, __m256i a, __m256i b);
VPBLENDMD __m128i __mm128i_mask_blend_epi32(__mmask8 m, __m128i a, __m128i b);
VPBLENDMQ __m512i __mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);
VPBLENDMQ __m256i __mm256_mask_blend_epi64(__mmask8 m, __m256i a, __m256i b);
VPBLENDMQ __m128i __mm128i_mask_blend_epi64(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E4.
VPBROADCASTB/W/D/Q—Load with Broadcast Integer Data from General Purpose Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 7A/r VPBROADCASTB xmm1 {k1}{z}, reg</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7A/r VPBROADCASTB ymm1 {k1}{z}, reg</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7A/r VPBROADCASTB zmm1 {k1}{z}, reg</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7B/r VPBROADCASTW xmm1 {k1}{z}, reg</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7B/r VPBROADCASTW ymm1 {k1}{z}, reg</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7B/r VPBROADCASTW zmm1 {k1}{z}, reg</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7C/r VPBROADCASTD xmm1 {k1}{z}, r32</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all double-words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7C/r VPBROADCASTD ymm1 {k1}{z}, r32</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all double-words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7C/r VPBROADCASTD zmm1 {k1}{z}, r32</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all double-words in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 7C/r VPBROADCASTQ xmm1 {k1}{z}, r64</td>
<td>T1S</td>
<td>V/N.E.</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quad-words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7C/r VPBROADCASTQ ymm1 {k1}{z}, r64</td>
<td>T1S</td>
<td>V/N.E.</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quad-words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7C/r VPBROADCASTQ zmm1 {k1}{z}, r64</td>
<td>T1S</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quad-words in the 512-bit destination subject to writemask k1.</td>
</tr>
</tbody>
</table>

NOTES:
1. EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

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

VPBROADCASTB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SRC[7:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+7:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VPBROADCASTW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SRC[15:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VPBROADCASTD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SRC[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VPBROADCASTQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
i ← j * 64
IF k1[j] OR *no writemask*
THEN DEST[i+63:i] ← SRC[63:0]
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] ← 0
FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VPBROADCASTB __m512i _mm512_mask_set1_epi8(__m512i s, __mmask64 k, int a);
VPBROADCASTB __m512i _mm512_maskz_set1_epi8( __mmask64 k, int a);
VPBROADCASTB __m256i _mm256_mask_set1_epi8(__m256i s, __mmask32 k, int a);
VPBROADCASTB __m256i _mm256_maskz_set1_epi8( __mmask32 k, int a);
VPBROADCASTB __m128i _mm_mask_set1_epi8(__m128i s, __mmask16 k, int a);
VPBROADCASTB __m128i _mm_maskz_set1_epi8( __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_mask_set1_epi32(__m512i s, __mmask16 k, __int64 a);
VPBROADCASTD __m512i _mm512_maskz_set1_epi32( __mmask16 k, __int64 a);
VPBROADCASTD __m256i _mm256_mask_set1_epi32(__m256i s, __mmask8 k, __int64 a);
VPBROADCASTD __m256i _mm256_maskz_set1_epi32( __mmask8 k, __int64 a);
VPBROADCASTD __m128i _mm_mask_set1_epi32(__m128i s, __mmask8 k, __int64 a);
VPBROADCASTD __m128i _mm_maskz_set1_epi32( __mmask8 k, __int64 a);
VPBROADCASTQ __m512i _mm512_mask_set1_epi64(__m512i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m512i _mm512_maskz_set1_epi64( __mmask8 k, __int64 a);
VPBROADCASTQ __m256i _mm256_mask_set1_epi64(__m256i s, __mmask16 k, __int64 a);
VPBROADCASTQ __m256i _mm256_maskz_set1_epi64( __mmask16 k, __int64 a);
VPBROADCASTQ __m128i _mm_mask_set1_epi64(__m128i s, __mmask16 k, __int64 a);
VPBROADCASTQ __m128i _mm_maskz_set1_epi64( __mmask16 k, __int64 a);
VPBROADCASTW __m512i _mm512_mask_set1_epi16(__m512i s, __mmask8 k, __int64 a);
VPBROADCASTW __m512i _mm512_maskz_set1_epi16( __mmask8 k, __int64 a);
VPBROADCASTW __m256i _mm256_mask_set1_epi16(__m256i s, __mmask16 k, __int64 a);
VPBROADCASTW __m256i _mm256_maskz_set1_epi16( __mmask16 k, __int64 a);
VPBROADCASTW __m128i _mm_mask_set1_epi16(__m128i s, __mmask16 k, __int64 a);
VPBROADCASTW __m128i _mm_maskz_set1_epi16( __mmask16 k, __int64 a);

Exceptions
EVEX-encoded instructions, see Exceptions Type E7NM.
#UD If EVEX.vvvv != 1111B.
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1, xmm2/m8</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a byte integer in the source operand to sixteen locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1, xmm2/m8</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a byte integer in the source operand to thirty-two locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1[1]{k1}{z}, xmm2/m8</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1[1]{k1}{z}, xmm2/m8</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 78 /r VPBROADCASTB zmm1[1]{k1}{z}, xmm2/m8</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1, xmm2/m16</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a word integer in the source operand to eight locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1, xmm2/m16</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a word integer in the source operand to sixteen locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1[1]{k1}{z}, xmm2/m16</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1[1]{k1}{z}, xmm2/m16</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 79 /r VPBROADCASTW zmm1[1]{k1}{z}, xmm2/m16</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1, xmm2/m32</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a dword integer in the source operand to four locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1, xmm2/m32</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a dword integer in the source operand to eight locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1[1]{k1}{z}, xmm2/m32</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1[1]{k1}{z}, xmm2/m32</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 58 /r VPBROADCASTD zmm1[1]{k1}{z}, xmm2/m32</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a qword element in source operand to two locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 59 /r VPBROADCASTQ ymm1, xmm2/m64</td>
<td>RM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast a qword element in source operand to four locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 59 /r VPBROADCASTQ xmm1[1]{k1}{z}, xmm2/m64</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 59 /r VPBROADCASTQ ymm1[1]{k1}{z}, xmm2/m64</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 59 /r VPBROADCASTQ zmm1[1]{k1}{z}, xmm2/m64</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 59 /r VPBROADCASTI32x2 xmm1[1]{k1}{z}, xmm2/m64</td>
<td>T2</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>
### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRMreg (w)</td>
<td>ModRMrmr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>T1S, T2, T4, T8</td>
<td>ModRMreg (w)</td>
<td>ModRMrmr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

**VEX256-encoded VPBROADCASTB/W/D/Q:** The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register. VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAX_VL-1:256) of the destination register are zeroed.

**EVEX-encoded VPBROADCASTSTD/Q:** The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

**VPBROADCASTI32X4 and VPBROADCASTI64X4:** The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VPBROADCASTI32X4 and VPBROADCASTI64X4 are reserved and will #UD.

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

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

Figure 5-17. VPBROADCASTD Operation (128-bit version)

Figure 5-18. VPBROADCASTQ Operation (256-bit version)

Figure 5-19. VBROADCASTI128 Operation (256-bit version)
VPBROADCAST—Load Integer and Broadcast

**Operation**

**VPBROADCASTB (EVEX encoded versions)**

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

FOR \(j \leftarrow 0\) TO \(KL-1\)

\[
i = j \times 8\]

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

THEN \(\text{DEST}[i+7:i] \leftarrow \text{SRC}[7:0]\)

ELSE

IF *merging-masking* ; merging-masking

THEN *\(\text{DEST}[i+7:i]\) remains unchanged*\n
ELSE ; zeroing-masking

\(\text{DEST}[i+7:i] \leftarrow 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAX}_V\text{L}-1:VL] \leftarrow 0\)

**VPBROADCASTW (EVEX encoded versions)**

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

FOR \(j \leftarrow 0\) TO \(KL-1\)

\[
i = j \times 16\]

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

THEN \(\text{DEST}[i+15:i] \leftarrow \text{SRC}[15:0]\)

ELSE

IF *merging-masking* ; merging-masking

THEN *\(\text{DEST}[i+15:i]\) remains unchanged*\n
ELSE ; zeroing-masking

\(\text{DEST}[i+15:i] \leftarrow 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAX}_V\text{L}-1:VL] \leftarrow 0\)

**VPBROADCASTD (128 bit version)**

\(\text{temp} \leftarrow \text{SRC}[31:0]\)

\(\text{DEST}[31:0] \leftarrow \text{temp}\)

\(\text{DEST}[63:32] \leftarrow \text{temp}\)

\(\text{DEST}[95:64] \leftarrow \text{temp}\)

\(\text{DEST}[127:96] \leftarrow \text{temp}\)

\(\text{DEST}[\text{MAX}_V\text{L}-1:128] \leftarrow 0\)
VPBROADCAST (VEX.256 encoded version)

temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
DEST[MAX_VL-1:256] ← 0

VPBROADCAST (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[31:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VPBROADCASTQ (VEX.256 encoded version)

temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAX_VL-1:256] ← 0

VPBROADCASTQ (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[63:0]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VBROADCASTI32x2 (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR \( j \leftarrow 0 \) TO \( KL-1 \)
\[
i \leftarrow j \times 32 \\
n \leftarrow (j \mod 2) \times 32 \\
\text{IF} \ k1[j] \text{ OR } \text{*no writemask*} \\
\quad \text{THEN} \ \text{DEST}[i+31:i] \leftarrow \text{SRC}[n+31:n] \\
\quad \text{ELSE} \\
\quad \quad \text{IF} \ *\text{merging-masking*} \ \ ; \ \text{merging-masking} \\
\quad \quad \quad \text{THEN} *\text{DEST}[i+31:i] \text{ remains unchanged*} \\
\quad \quad \quad \text{ELSE} \ \ ; \ \text{zeroing-masking} \\
\quad \quad \quad \quad \text{DEST}[i+31:i] \leftarrow 0 \\
\quad \text{FI} \\
\text{FI}; \ \text{ENDFOR}
\]
\( \text{DEST}[\text{MAX}_V L - 1 : V L] \leftarrow 0 \)

**VBROADCAST128 (VEX.256 encoded version)**
\[
\text{temp} \leftarrow \text{SRC}[127:0] \\
\text{DEST}[127:0] \leftarrow \text{temp} \\
\text{DEST}[255:128] \leftarrow \text{temp} \\
\text{DEST}[\text{MAX}_V L - 1 : 256] \leftarrow 0
\]

**VBROADCAST32X4 (EVEX encoded versions)**
\((KL, VL) = (8, 256), (16, 512)\)
\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
i \leftarrow j \times 32 \\
n \leftarrow (j \mod 4) \times 32 \\
\text{IF} \ k1[j] \text{ OR } \text{*no writemask*} \\
\quad \text{THEN} \ \text{DEST}[i+31:i] \leftarrow \text{SRC}[n+31:n] \\
\quad \text{ELSE} \\
\quad \quad \text{IF} \ *\text{merging-masking*} \ \ ; \ \text{merging-masking} \\
\quad \quad \quad \text{THEN} *\text{DEST}[i+31:i] \text{ remains unchanged*} \\
\quad \quad \quad \text{ELSE} \ \ ; \ \text{zeroing-masking} \\
\quad \quad \quad \quad \text{DEST}[i+31:i] \leftarrow 0 \\
\quad \text{FI} \\
\text{FI}; \ \text{ENDFOR}
\]
\( \text{DEST}[\text{MAX}_V L - 1 : V L] \leftarrow 0 \)

**VBROADCAST64X2 (EVEX encoded versions)**
\((KL, VL) = (8, 256), (16, 512)\)
\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
i \leftarrow j \times 64 \\
n \leftarrow (j \mod 2) \times 64 \\
\text{IF} \ k1[j] \text{ OR } \text{*no writemask*} \\
\quad \text{THEN} \ \text{DEST}[i+63:i] \leftarrow \text{SRC}[n+63:n] \\
\quad \text{ELSE} \\
\quad \quad \text{IF} \ *\text{merging-masking*} \ \ ; \ \text{merging-masking} \\
\quad \quad \quad \text{THEN} *\text{DEST}[i+63:i] \text{ remains unchanged*} \\
\quad \quad \quad \text{ELSE} \ \ ; \ \text{zeroing-masking} \\
\quad \quad \quad \quad \text{DEST}[i+63:i] = 0 \\
\quad \text{FI} \\
\text{FI}; \ \text{ENDFOR}
VPBROADCASTI32X8 (EVEX.U1.512 encoded version)

FOR $j \leftarrow 0$ TO $15$
  $i \leftarrow j \times 32$
  $n \leftarrow (j \mod 8) \times 32$
  IF $k1[j] \text{ OR } \text{no writemask}$
    THEN $\text{DEST}[i+31:i] \leftarrow \text{SRC}[n+31:n]$
  ELSE
    IF *merging-masking* ; merging-masking
       THEN *DEST*[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
       $\text{DEST}[i+31:i] \leftarrow 0$
    FI
  FI
ENDFOR
$\text{DEST}[\text{MAX}_V\text{L}-1:V\text{L}] \leftarrow 0$

VPBROADCASTI64X4 (EVEX.512 encoded version)

FOR $j \leftarrow 0$ TO $7$
  $i \leftarrow j \times 64$
  $n \leftarrow (j \mod 4) \times 64$
  IF $k1[j] \text{ OR } \text{no writemask}$
    THEN $\text{DEST}[i+63:i] \leftarrow \text{SRC}[n+63:n]$
  ELSE
    IF *merging-masking* ; merging-masking
       THEN *DEST*[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
       $\text{DEST}[i+63:i] \leftarrow 0$
    FI
  FI
ENDFOR
$\text{DEST}[\text{MAX}_V\text{L}-1:V\text{L}] \leftarrow 0$

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCAST _m512i _mm512_broadcastb_epi8( _m128i a);
VPBROADCAST _m512i _mm512_mask_broadcastb_epi8( _m512i s, _mmask64 k, _m128i a);
VPBROADCAST _m512i _mm512_maskz_broadcastb_epi8( _mmask64 k, _m128i a);
VPBROADCAST _m256i _mm256_broadcastb_epi8( _m128i a);
VPBROADCAST _m256i _mm256_mask_broadcastb_epi8( _mmask32 k, _m128i a);
VPBROADCAST _m256i _mm256_maskz_broadcastb_epi8( _mmask32 k, _m128i a);
VPBROADCAST _m128i _mm_mask_broadcastb_epi8( _mmask16 k, _m128i a);
VPBROADCAST _m128i _mm_maskz_broadcastb_epi8( _mmask16 k, _m128i a);
VPBROADCAST _m128i _mm_broadcastb_epi8( _m128i a);
VPBROADCASTD _m512i _mm512_broadcastd_epi32( _m128i a);
VPBROADCASTD _m512i _mm512_mask_broadcastd_epi32( _m512i s, _mmask16 k, _m128i a);
VPBROADCASTD _m512i _mm512_maskz_broadcastd_epi32( _mmask16 k, _m128i a);
VPBROADCASTD _m256i _mm256_broadcastd_epi32( _m128i a);
VPBROADCASTD _m256i _mm256_mask_broadcastd_epi32( _mmask8 k, _m128i a);
VPBROADCASTD _m256i _mm256_maskz_broadcastd_epi32( _mmask8 k, _m128i a);
VPBROADCASTD _m128i _mmBroadcastd_epi32( _m128i a);
VPBROADCASTD _m128i _mm_mask_broadcastd_epi32( _mmask8 k, _m128i a);
VPBROADCASTD _m128i _mm_maskz_broadcastd_epi32( _mmask8 k, _m128i a);
VPBROADCASTD _m128i _mm_broadcastd_epi32( _m128i a);
VPBROADCASTD _mm512_broadcastq_epi64( _m128i a);
VPBROADCASTD _mm512_maskq_broadcastq_epi64( _m512i s, _mmask8 k, _m128i a);
VPBROADCASTD _mm512_maskz_broadcastq_epi64( _mmask8 k, _m128i a);
VPBROADCASTQ __m256i _mm256_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m256i _mm256_mask_broadcastq_epi64(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m128i _mm_mask_broadcastq_epi64(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_maskz_broadcastq_epi64(__mmask8 k, __m128i a);

VPBROADCASTW __m512i _mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i _mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCASTW __m512i _mm512_maskz_broadcastw_epi16(__mmask32 k, __m128i a);
VPBROADCASTW __m256i _mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i _mm256_mask_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCASTW __m256i _mm256_maskz_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCASTW __m128i _mm_broadcastw_epi16(__m128i a);
VPBROADCASTW __m128i _mm_mask_broadcastw_epi16(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTW __m128i _mm_maskz_broadcastw_epi16(__mmask8 k, __m128i a);

VBROADCASTI32x2 __m512i _mm512_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m512i _mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBroadcastI32x2 __m512i _mm512_maskz_broadcast_i32x2(__mmask16 k, __m128i a);
VBroadcastI32x2 __m256i _mm256_broadcast_i32x2(__m128i a);
VBroadcastI32x2 __m256i _mm256_maskroadcast_i32x2(__m256i s, __mmask8 k, __m128i a);
VBroadcastI32x2 __m256i _mm256_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBroadcastI32x2 __m128i _mm_broadcastq_i32x2(__m128i a);
VBroadcastI32x2 __m128i _mm_mask_broadcastq_i32x2(__m128i s, __mmask8 k, __m128i a);
VBroadcastI32x2 __m128i _mm_maskz_broadcastq_i32x2(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, syntax with reg/mem operand, see Exceptions Type E6.

#UD If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.
If EVEX.L’L = 0 for VbroadcastI32x4/VbroadcastI64x2.
If EVEX.L’L < 10b for VbroadcastI32x8/VbroadcastI64x4.
VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 3F /r lb</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 3F /r lb</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 3F /r lb</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 3E /r lb</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 3E /r lb</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 3E /r lb</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed unsigned byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVM</td>
<td>ModRMreg (w)</td>
<td>vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.
The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.

### Table 5-10. Pseudo-Op and VPCMP* Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>PCMPM Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPCMPEQ* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VPCMPLT* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VPCMPLE* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VPCMPNEQ* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VPPCPMPNLT* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VPCMPNLE* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 6</td>
</tr>
</tbody>
</table>

**Operation**

CASE (COMPARISON PREDICATE) OF

- 0: OP ← EQ;
- 1: OP ← LT;
- 2: OP ← LE;
- 3: OP ← FALSE;
- 4: OP ← NEQ;
- 5: OP ← NLT;
- 6: OP ← NLE;
- 7: OP ← TRUE;

ESAC;

**VPCMPB (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j * 8

IF k2[j] OR *no writemask*

THEN

CMP ← SRC1[i+7:i] OP SRC2[i+7:i];

IF CMP = TRUE

THEN DEST[j] ← 1;
ELSE DEST[j] ← 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX_KL-1:KL] ← 0
VPCMPUB (EVEX encoded versions)

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

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            CMP ← SRC1[i+7:i] OP SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE DEST[j] = 0 ; zeroin masking onlyFI;
    FI;
ENDFOR

DEST[MAX_KL-1:KL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPB __mmask64 __mm512_cmp_epi8_mask(__m512i a, __m512i b, int cmp);
VPCMPB __mmask64 __mm512_mask_cmp_epi8_mask(__mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPB __mmask32 __mm256_cmp_epi8_mask(__m256i a, __m256i b, int cmp);
VPCMPB __mmask32 __mm256_mask_cmp_epi8_mask(__mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPB __mmask16 __mm_cmp_epi8_mask(__m128i a, __m128i b, int cmp);
VPCMPB __mmask16 __mm_mask_cmp_epi8_mask(__mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPB __mmask64 __mm512_cmp_eq_epi8_mask(__m512i a, __m512i b);
VPCMPB __mmask64 __mm512_mask_cmp_eq_epi8_mask(__mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32 __mm256_cmp_eq_epi8_mask(__m256i a, __m256i b);
VPCMPB __mmask32 __mm256_mask_cmp_eq_epi8_mask(__mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16 __mm_cmp_eq_epi8_mask(__m128i a, __m128i b);
VPCMPB __mmask16 __mm_mask_cmp_eq_epi8_mask(__mmask16 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.
VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 1F / r lb</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 1F / r lb</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 1F / r lb</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 1E / r lb</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 1E / r lb</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 1E / r lb</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPD/VPCMPUD performs a comparison between pairs of signed/unsigned doubleword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register k1. Up to 16/8/4 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.
Operation

CASE (COMPARISON PREDICATE) OF
  0: OP \leftarrow EQ;
  1: OP \leftarrow LT;
  2: OP \leftarrow LE;
  3: OP \leftarrow FALSE;
  4: OP \leftarrow NEQ;
  5: OP \leftarrow NLT;
  6: OP \leftarrow NLE;
  7: OP \leftarrow TRUE;
ESAC;

VPCMPD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j \leftarrow 0 TO KL-1
  i \leftarrow j * 32
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP \leftarrow SRC1[i+31:i] \text{ OP SRC2}[31:0];
        ELSE CMP \leftarrow SRC1[i+31:i] \text{ OP SRC2}[i+31:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] \leftarrow 1;
        ELSE DEST[j] \leftarrow 0; FI;
    ELSE DEST[j] \leftarrow 0 ; zeroing-masking onlyFI;
    FI;
  ENDFOR
DEST[MAX_KL-1:KL] \leftarrow 0

VPCMPUD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j \leftarrow 0 TO KL-1
  i \leftarrow j * 32
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP \leftarrow SRC1[i+31:i] \text{ OP SRC2}[31:0];
        ELSE CMP \leftarrow SRC1[i+31:i] \text{ OP SRC2}[i+31:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] \leftarrow 1;
        ELSE DEST[j] \leftarrow 0; FI;
    ELSE DEST[j] \leftarrow 0 ; zeroing-masking onlyFI;
    FI;
  ENDFOR
DEST[MAX_KL-1:KL] \leftarrow 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCMPD __mmask16 __mm512_cmp_epi32_mask(__m512i a, __m512i b, int imm);
VPCMPD __mmask16 __mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m512i a, __m512i b);
VPCMPD __mmask16 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPDUD __mmask16 __mm512_cmp_epi32_mask( __m512i a, __m512i b, int imm);
VPCMPDUD __mmask16 __mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask8 __mm256_cmp_epi32_mask( __m256i a, __m256i b, int imm);
VPCMPD __mmask8 __mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 __mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m256i a, __m256i b);
VPCMPD __mmask8 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 __mm256_cmp_epi32_mask( __m256i a, __m256i b, int imm);
VPCMPD __mmask8 __mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 __mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m256i a, __m256i b);
VPCMPD __mmask8 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 __mm512_cmp_epi32_mask( __m512i a, __m512i b, int imm);
VPCMPD __mmask8 __mm512_mask_cmp_epi32_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask8 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m512i a, __m512i b);
VPCMPD __mmask8 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPD __mmask8 __mm128_cmp_epi32_mask( __m128i a, __m128i b, int imm);
VPCMPD __mmask8 __mm128_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPD __mmask8 __mm128_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m128i a, __m128i b);
VPCMPD __mmask8 __mm128_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.
**VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>FV/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>FV/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed signed quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>FV/V</td>
<td>AVX512F</td>
<td>Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>FV/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>FV/V</td>
<td>AVX512VL AVX512F</td>
<td>Compare packed unsigned quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>FV/V</td>
<td>AVX512F</td>
<td>Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.
Operation

CASE (COMPARISON PREDICATE) OF
  0: OP ← EQ;
  1: OP ← LT;
  2: OP ← LE;
  3: OP ← FALSE;
  4: OP ← NEQ;
  5: OP ← NLT;
  6: OP ← NLE;
  7: OP ← TRUE;
ESAC;

VPCMPQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];
        ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPCMPUQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];
        ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCMPQ __mmask8 __mm512_cmp_epi64_mask( __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 __mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m512i a, __m512i b);
VPCMPQ __mmask8 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPQ __mmask8 __mm512_cmp_epi64_mask( __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 __mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 __mm512_cmp_epu64_mask( __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 __mm512_mask_cmp_epu64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 __mm512_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m512i a, __m512i b);
VPCMPQ __mmask8 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPQ __mmask8 __mm256_cmp_epi64_mask( __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 __mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 __mm256_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m256i a, __m256i b);
VPCMPQ __mmask8 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPQ __mmask8 __mm256_cmp_epi64_mask( __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 __mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 __mm256_cmp_epu64_mask( __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 __mm256_mask_cmp_epu64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 __mm256_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m256i a, __m256i b);
VPCMPQ __mmask8 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPQ __mmask8 __mm128_cmp_epi64_mask( __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 __mm128_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 __mm128_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m128i a, __m128i b);
VPCMPQ __mmask8 __mm128_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPQ __mmask8 __mm128_cmp_epi64_mask( __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 __mm128_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 __mm128_cmp_epu64_mask( __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 __mm128_mask_cmp_epu64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 __mm128_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m128i a, __m128i b);
VPCMPQ __mmask8 __mm128_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.
VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVM</td>
<td>ModRMreg (w)</td>
<td>vvvv (r)</td>
<td>ModRMtr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.
Operation

CASE (COMPARISON PREDICATE) OF
  0: OP ← EQ;
  1: OP ← LT;
  2: OP ← LE;
  3: OP ← FALSE;
  4: OP ← NEQ ;
  5: OP ← NLT;
  6: OP ← NLE;
  7: OP ← TRUE;
ESAC;

VPCMPw (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k2[j] OR *no writemask*
    THEN
      ICMP ← SRC1[i+15:i] OP SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] = 0 ; zeroing-masking only
      FI;
  ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPCMPUw (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k2[j] OR *no writemask*
    THEN
      CMP ← SRC1[i+15:i] OP SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] ← 1;
        ELSE DEST[j] ← 0; FI;
      ELSE DEST[j] = 0 ; zeroing-masking only
      FI;
  ENDFOR
DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCMPW __mmask32 __mm512_cmp_epi16_mask( __m512i a, __m512i b, int cmp);
VPCMPW __mmask32 __mm512_mask_cmp_epi16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPW __mmask16 __mm256_cmp_epi16_mask( __mm256i a, __mm256i b, int cmp);
VPCMPW __mmask16 __mm256_mask_cmp_epi16_mask( __mmask16 m, __mm256i a, __mm256i b, int cmp);
VPCMPW __mmask8 __mm_cmp_epi16_mask( __mm128i a, __mm128i b, int cmp);
VPCMPW __mmask8 __mm_mask_cmp_epi16_mask( __mmask8 m, __mm128i a, __mm128i b, int cmp);
VPCMPW __mmask32 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __m512i a, __m512i b);
VPCMPW __mmask32 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 __mm256_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mm256i a, __mm256i b);
VPCMPW __mmask16 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask16 m, __mm256i a, __mm256i b);
VPCMPW __mmask8 __mm_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mm128i a, __mm128i b);
VPCMPW __mmask8 __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask( __mmask8 m, __mm128i a, __mm128i b);
VPCMPUW __mmask32 __mm512_cmp_epu16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 __mm512_mask_cmp_epu16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 __mm256_cmp_epu16_mask( __m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 __mm256_mask_cmp_epu16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 __mm_cmp_epu16_mask( __m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 __mm_mask_cmp_epu16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPUW __mmask32 __mm512_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 __mm256_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mm256i a, __mm256i b, int cmp);
VPCMPUW __mmask16 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask16 m, __mm256i a, __mm256i b, int cmp);
VPCMPUW __mmask8 __mm_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mm128i a, __mm128i b, int cmp);
VPCMPUW __mmask8 __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask8 m, __mm128i a, __mm128i b, int cmp);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.
VPcompressd—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 8B /r VPCOMPRESSD xmm1/m128 [k1]{z}, xmm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed doubleword integer values from xmm2 to xmm1/m128 using controlmask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8B /r VPCOMPRESSD ymm1/m256 [k1]{z}, ymm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed doubleword integer values from ymm2 to ymm1/m256 using controlmask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 [k1]{z}, zmm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed doubleword integer values from zmm2 to zmm1/m512 using controlmask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:rr/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

**VPcompressd (EVEX encoded versions) store form**

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

\[\text{SIZE} \leftarrow 32\]

\[k \leftarrow 0\]

FOR \(j \leftarrow 0 \text{ TO } KL-1\)

\[i \leftarrow j \times 32\]

IF \(k1[j] \text{ OR } \text{no controlmask}\)

THEN

\[\text{DEST}[k+\text{SIZE}-1:k] \leftarrow \text{SRC}[i+31:j]\]

\[k \leftarrow k + \text{SIZE}\]

FI;

ENDFOR;
VPCOMPRESSD (EVEX encoded versions) reg-reg form

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

SIZE ≜ 32

k ≜ 0

FOR j ≜ 0 TO KL-1
    i ≜ j * 32
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] ← SRC[i+31:i]
            k ≜ k + SIZE
        FI;
    ENDFOR

IF *merging-masking*
    THEN *DEST[VL-1:k] remains unchanged*
    ELSE DEST[VL-1:k] ← 0
FI

DEST[MAX_VL-1:VL] ≜ 0

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSD __m512i _mm512_mask_compress_epi32(__m512i s, __mmask16 c, __m512i a);
VPCOMPRESSD __m512i _mm512_maskz_compress_epi32(__mmask16 c, __m512i a);
VPCOMPRESSD void _mm512_mask_compressstoreu_epi32(void * a, __mmask16 c, __m512i s);
VPCOMPRESSD __m256i _mm256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i _mm256_maskz_compress_epi32(__mmask8 c, __m256i a);
VPCOMPRESSD void _mm256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i _mm_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i _mm_maskz_compress_epi32(__mmask8 c, __m128i a);
VPCOMPRESSD void _mm_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E4.nb.
VPCOMPRESSIONQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 8B /r VPCOMPRESSIONQ xmm1/m128 (k1)[z], xmm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed quadword integer values from xmm2 to xmm1/m128 using controlmask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 8B /r VPCOMPRESSIONQ ymm1/m256 (k1)[z], ymm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed quadword integer values from ymm2 to ymm1/m256 using controlmask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 8B /r VPCOMPRESSIONQ zmm1/m512 (k1)[z], zmm2</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed quadword integer values from zmm2 to zmm1/m512 using controlmask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSIONQ (EVEX encoded versions) store form

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

SIZE ← 64
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] ← SRC[i+63:i]
            k ← k + SIZE
    FI;
ENFOR
VPCOMPRESSQ (EVEX encoded versions) reg-reg form

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

\[SIZE \leftarrow 64\]

\[k \leftarrow 0\]

FOR \[j \leftarrow 0\] TO \[KL-1\]

\[i \leftarrow j \times 64\]

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

THEN

\[DEST[k+SIZE-1:k] \leftarrow SRC[i+63:i]\]

\[k \leftarrow k + SIZE\]

ENDIF

ENDFOR

IF *merging-masking*

THEN *DEST[VL-1:k] remains unchanged*

ELSE \[DEST[VL-1:k] \leftarrow 0\]

FI

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

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSQ __m512i _mm512_mask_compress_epi64(__m512i s, __mmask8 c, __m512i a);
VPCOMPRESSQ __m512i _mm512_maskz_compress_epi64(__mmask8 c, __m512i a);
VPCOMPRESSQ void _mm512_mask_compressstoreu_epi64(void * a, __mmask8 c, __m512i s);
VPCOMPRESSQ __m256i _mm256_mask_compress_epi64(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSQ __m256i _mm256_maskz_compress_epi64(__mmask8 c, __m256i a);
VPCOMPRESSQ void _mm256_mask_compressstoreu_epi64(void * a, __mmask8 c, __m256i s);
VPCOMPRESSQ __m128i _mm_mask_compress_epi64(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSQ __m128i _mm_maskz_compress_epi64(__mmask8 c, __m128i a);
VPCOMPRESSQ void _mm_mask_compressstoreu_epi64(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.
VPCONFICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/ Register

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bitMode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 C4 /r VPCONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 C4 /r VPCONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 C4 /r VPCONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 C4 /r VPCONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element’s comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

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

VPCONFLICTD

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    IF MaskBit(j) OR *no writemask* THEN
        FOR k ← 0 TO j-1
            m ← k*32
            IF ((SRC[i+31:i] = SRC[m+31:m])) THEN
                DEST[i+k] ← 1
            ELSE
                DEST[i+k] ← 0
            FI
        ENDFOR
        DEST[i+31:i+j] ← 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+31:i] remains unchanged*
        ELSE
            DEST[i+31:i] ← 0
        FI
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VPCONFLICTQ

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j*64
    IF MaskBit(j) OR *no writemask* THEN
        FOR k ← 0 TO j-1
            m ← k*64
            IF ((SRC[i+63:i] = SRC[m+63:m])) THEN
                DEST[i+k] ← 1
            ELSE
                DEST[i+k] ← 0
            FI
        ENDFOR
        DEST[i+63:i+j] ← 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] ← 0
        FI
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCONFICTD __m512i __m512_conflict_epi32(__m512i a);
VPCONFICTD __m512i __m512_mask_conflict_epi32(__m512i s, __mmask16 m, __m512i a);
VPCONFICTD __m512i __m512_maskz_conflict_epi32(__mmask16 m, __m512i a);
VPCONFICTQ __m512i __m512_conflict_epi64(__m512i a);
VPCONFICTQ __m512i __m512_mask_conflict_epi64(__m512i s, __mmask8 m, __m512i a);
VPCONFICTQ __m512i __m512_maskz_conflict_epi64(__mmask8 m, __m512i a);

VPCONFICTD __m256i __m256_conflict_epi32(__m256i a);
VPCONFICTD __m256i __m256_mask_conflict_epi32(__m256i s, __mmask8 m, __m256i a);
VPCONFICTD __m256i __m256_maskz_conflict_epi32(__mmask8 m, __m256i a);

VPCONFICTQ __m256i __m256_conflict_epi64(__m256i a);
VPCONFICTQ __m256i __m256_mask_conflict_epi64(__m256i s, __mmask8 m, __m256i a);
VPCONFICTQ __m256i __m256_maskz_conflict_epi64(__mmask8 m, __m256i a);

VPCONFICTD __m128i __m128_conflict_epi32(__m128i a);
VPCONFICTD __m128i __m128_mask_conflict_epi32(__m128i s, __mmask8 m, __m128i a);
VPCONFICTD __m128i __m128_maskz_conflict_epi32(__mmask8 m, __m128i a);

VPCONFICTQ __m128i __m128_conflict_epi64(__m128i a);
VPCONFICTQ __m128i __m128_mask_conflict_epi64(__m128i s, __mmask8 m, __m128i a);
VPCONFICTQ __m128i __m128_maskz_conflict_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.
**VPERM2F128 — Permute Floating-Point Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.0F3A.W0 06 /r ib</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.</td>
</tr>
<tr>
<td>VPERM2F128 ymm1, ymm2, ymm3/m256, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

**Figure 5-21. VPERM2F128 Operation**

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.
Operation

VPERM2F128

CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
2: DEST[255:128] ← SRC2[127:0]
ESAC

IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[VL MAX-1:128] ← 0
FI

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128: __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)
VPERM2F128: __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)
VPERM2F128: __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD  If VEX.L = 0
                       If VEX.W = 1.
VPERM2I128 — Permute Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.0F3A.W0 46/r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permute 128-bit integer data in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.</td>
</tr>
<tr>
<td>VPERM2I128 ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVMI</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Permute 128 bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

**Figure 5-22. VPERM2I128 Operation**

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.
Operation

VPERM2I128
case imm8[1:0] of
0: dest[127:0] <=> src1[127:0]
1: dest[127:0] <=> src1[255:128]
2: dest[127:0] <=> src2[127:0]
esac
case imm8[5:4] of
0: dest[255:128] <=> src1[127:0]
2: dest[255:128] <=> src2[127:0]
esac
if (imm8[3])
dest[127:0] <= 0
fi
if (imm8[7])
dest[255:128] <= 0
fi

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2I128: __m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD
  if VEX.L = 0,
  if VEX.W = 1.
### VPERMD/VPERMW—Permute Packed Doublewords/Words Elements

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 36 /r VPERMD zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 8D /r VPERMW xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 8D /r VPERMW ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers in ymm3/m256 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 8D /r VPERMW zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FVM</td>
<td>ModRMreg (w)</td>
<td>VEX.vvvv</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

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

**VEX.256 encoded VPERMD:** The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAX_VL-1:256) of the corresponding destination register are zeroed.

**EVEX encoded VPERMD:** The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

**VPERMW:** first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

**EVEX.128 encoded versions:** Bits (MAX_VL-1:128) of the corresponding ZMM register are zeroed.
Operation

**VPERMD (EVEX encoded versions)**

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

IF VL = 256 THEN n ← 2; Fl;

IF VL = 512 THEN n ← 3; Fl;

FOR j ← 0 TO KL-1

    i ← j * 32
    id ← 32*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC2[31:0];
                ELSE DEST[i+31:i] ← SRC2[id+31:id];
                Fl;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+31:i] ← 0
                Fl
        FI
    FI;

ENDFOR

DEST[MAX_VL-1:VL] ← 0

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

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];

DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];

DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];

DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];

DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];

DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];

DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];

DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];

DEST[MAX_VL-1:256] ← 0

**VPERMW (EVEX encoded versions)**

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

IF VL = 128 THEN n ← 2; Fl;

IF VL = 256 THEN n ← 3; Fl;

IF VL = 512 THEN n ← 4; Fl;

FOR j ← 0 TO KL-1

    i ← j * 16
    id ← 16*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC2[id+15:id]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+15:i] ← 0
            Fl
        FI
    FI;

ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPERMD __m512i _mm512_permutexvar_epi32( __m512i idx, __m512i a);
VPERMD __m512i _mm512_mask_permutexvar_epi32( __m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i _mm512_maskz_permutexvar_epi32( __mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i _mm256_permutexvar_epi32( __m256i idx, __m256i a);
VPERMD __m256i _mm256_mask_permutexvar_epi32( __m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i _mm256_maskz_permutexvar_epi32( __mmask8 k, __m256i idx, __m256i a);

VPERMW __m512i _mm512_permutexvar_epi16( __m512i idx, __m512i a);
VPERMW __m512i _mm512_mask_permutexvar_epi16( __m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i _mm512_maskz_permutexvar_epi16( __mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i _mm256_permutexvar_epi16( __m256i idx, __m256i a);
VPERMW __m256i _mm256_mask_permutexvar_epi16( __m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i _mm256_maskz_permutexvar_epi16( __mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i _mm_permutexvar_epi16( __m128i idx, __m128i a);
VPERMW __m128i _mm_mask_permutexvar_epi16( __m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i _mm_maskz_permutexvar_epi16( __mmask8 k, __m128i idx, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPERMD, see Exceptions Type E4NF.
EVEX-encoded VPERMW, see Exceptions Type E4NF.nb.

#UD If VEX.L = 0.
If EVEX.L'L = 0 for VPERMD.
### VPERM2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 75 /r VPERM2W xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 75 /r VPERM2W ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W1 75 /r VPERM2W zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 76 /r VPERM2D xmm1 [k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 76 /r VPERM2D ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 76 /r VPERM2D zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 76 /r VPERM2Q xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 76 /r VPERM2Q ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W1 76 /r VPERM2Q zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F38.W0 77 /r VPERM2PS xmm1 [k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W0 77 /r VPERM2PS ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W0 77 /r VPERM2PS zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
VPERM2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

In the instruction set reference, V-Z

Description
Permutes 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1. The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2). Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

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

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

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

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

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F38.W1 77 /r VPERM2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F38.W1 77 /r VPERM2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F38.W1 77 /r VPERM2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVM</td>
<td>ModRM:reg (r,w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
**Operation**

**VPERMI2W (EVEX encoded versions)**

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

IF VL = 128
   id ← 2
   FI;
IF VL = 256
   id ← 3
   FI;
IF VL = 512
   id ← 4
   FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1
   i ← j * 16
   off ← 16*TMP_DEST[i+id]
   IF k1[j] OR *no writemask*
       THEN
       DEST[i+15:i] ← TMP_DEST[i+id+1] ? SRC2[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

**VPERMI2D/VPERMI2PS (EVEX encoded versions)**

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

IF VL = 128
   id ← 1
   FI;
IF VL = 256
   id ← 2
   FI;
IF VL = 512
   id ← 3
   FI;
TMP_DEST ← DEST
FOR j ← 0 TO KL-1
   i ← j * 32
   off ← 32*TMP_DEST[i+id]
   IF k1[j] OR *no writemask*
       THEN
       IF (EVEX.b = 1) AND (SRC2 *is memory*)
           THEN
           DEST[i+31:i] ← TMP_DEST[i+id+1] ? SRC2[31:0]
           ELSE
           DEST[i+31:i] ← TMP_DEST[i+id+1] ? SRC2[off+31:off]
           : SRC1[off+31:off]
VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

INSTRUCTION SET REFERENCE, V-Z

5-368 Vol. 2C

ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+31:i] \( \leftarrow 0 \)
FI
FI
ENDFOR
DEST[MAX_VL-1:VL] \( \leftarrow 0 \)

VPERMI2Q/VPERMI2PD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
id \( \leftarrow 0 \)
FI;
IF VL = 256
id \( \leftarrow 1 \)
FI;
IF VL = 512
id \( \leftarrow 2 \)
FI;
TMP_DEST \( \leftarrow \) DEST
FOR j \( \leftarrow 0 \) TO KL-1
i \( \leftarrow j \times 64 \)
off \( \leftarrow 64 \times \text{TMP_DEST}[i+id] \)
IF k1[j] OR *no writemask*
THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN
ELSE
FI
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] \( \leftarrow 0 \)
FI
FI
ENDFOR
DEST[MAX_VL-1:VL] \( \leftarrow 0 \)
Intel C/C++ Compiler Intrinsic Equivalent

VPERM2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERM2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERM2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERM2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERM1 __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERM2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERM2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERM2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERM2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERM2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERM2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERM2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERM2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERM2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERM2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERM2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERM2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERM2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERM2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERM2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERM2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERM2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERM2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERM2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERM2PD __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERM2PD __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERM2PD __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERM2PD __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERM2PD __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERM2PD __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERM2PD __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERM2PD __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERM2PD __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERM2PD __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERM2PD __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERM2PD __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERM2PD __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERM2PD __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERM2PD __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERM2PD __m512i _mm512_maskz_permutex2var_epi64(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERM2PD __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERM2PD __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERM2PD __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERM2PD __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERM2PD __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERM2PD __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERM2PD __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERM2PD __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERM2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERM2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERM2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERM2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERM2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERM2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERM2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERM2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERM2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERM2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERM2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERM2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
VPERM2D/Q/PS/PD: See Exceptions Type E4NF.
VPERM2W: See Exceptions Type E4NF.nb.
### VPERMILPD—Permute In-Lane of Pairs of Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 0D /r</td>
<td>RVM V/V</td>
<td>AVX</td>
<td>Permute double-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 0D /r</td>
<td>RVM V/V</td>
<td>AVX</td>
<td>Permute double-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 0D /r</td>
<td>FV-RVM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 0D /r</td>
<td>FV-RVM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 0D /r</td>
<td>FV-RVM V/V</td>
<td>AVX512F</td>
<td>Permute double-precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 05 /ib</td>
<td>RM V/V</td>
<td>AVX</td>
<td>Permute double-precision floating-point values in xmm2/m128 using controls from imm8.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 05 /ib</td>
<td>RM V/V</td>
<td>AVX</td>
<td>Permute double-precision floating-point values in ymm2/m256 using controls from imm8.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 05 /ib</td>
<td>FV-RM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 05 /ib</td>
<td>FV-RM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 05 /ib</td>
<td>FV-RM V/V</td>
<td>AVX512F</td>
<td>Permute double-precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
(variable control version)
Permute pairs of double-precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permuted results are stored in the destination operand (first operand).

The control bits are located at bit 0 of each quadword element (see Figure 5-24). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.

VEX.256 encoded version: Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

(immediate control version)
Permute pairs of double-precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.
Operation

VPERMILPD (EVEX immediate versions)

KL, VL = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC1 *is memory*)
    THEN TMP_SRC1[i+63:i] ← SRC1[63:0];
    ELSE TMP_SRC1[i+63:i] ← SRC1[i+63:i];
  FI;
ENDFOR;
IF (imm8[0] = 0) THEN TMP_DEST[63:0] ← SRC1[63:0]; FI;
IF (imm8[0] = 1) THEN TMP_DEST[63:0] ← TMP_SRC1[127:64]; FI;
IF (imm8[1] = 0) THEN TMP_DEST[127:64] ← TMP_SRC1[63:0]; FI;
IF (imm8[1] = 1) THEN TMP_DEST[127:64] ← TMP_SRC1[127:64]; FI;
IF VL >= 256
FI;
IF VL >= 512
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-mask
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-mask
      DEST[i+63:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VPERMILPD (256-bit immediate version)

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
DEST[MAX_VL-1:256] ← 0
VPERMILPD (128-bit immediate version)
IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
DEST[MAX_VL-1:128] ← 0

VPERMILPD (EVEX variable versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
  FI;
ENDFOR;
IF (TMP_SRC2[1] = 0) THEN TMP_DEST[63:0] ← SRC1[63:0]; FI;
IF (TMP_SRC2[1] = 1) THEN TMP_DEST[63:0] ← SRC1[127:64]; FI;
IF (TMP_SRC2[65] = 0) THEN TMP_DEST[127:64] ← SRC1[63:0]; FI;
IF (TMP_SRC2[65] = 1) THEN TMP_DEST[127:64] ← SRC1[127:64]; FI;
IF VL ≥ 256
  IF (TMP_SRC2[193] = 0) THEN TMP_DEST[255:192] ← SRC1[191:128]; FI;
FI;
IF VL ≥ 512
  IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] ← SRC1[319:256]; FI;
  IF (TMP SRC2[257] = 1) THEN TMP_DEST[319:256] ← SRC1[383:320]; FI;
  IF (TMP_SRC2[321] = 0) THEN TMP_DEST[383:320] ← SRC1[319:256]; FI;
  IF (TMP_SRC2[385] = 0) THEN TMP_DEST[447:384] ← SRC1[447:384]; FI;
  IF (TMP_SRC2[385] = 1) THEN TMP_DEST[447:384] ← SRC1[511:448]; FI;
  IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] ← SRC1[447:384]; FI;
  IF (TMP_SRC2[449] = 1) THEN TMP_DEST[511:448] ← SRC1[511:448]; FI;
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI
  ENDFOR
DEST[MAX_VL-1:VL] ← 0
VPERMILPD (256-bit variable version)

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
IF (SRC2[193] = 0) THEN DEST[255:192] ← SRC1[255:192]
DEST[MAX_VL-1:256] ← 0

VPERMILPD (128-bit variable version)

IF (SRC2[1] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPD __m512d _mm512_permute_pd( __m512d a, int imm);
VPERMILPD __m512d _mm512_mask_permute_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d _mm512_maskz_permute_pd( __mmask8 k, __m512d a, int imm);
VPERMILPD __m256d _mm256_permute_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d _mm256_mask_permute_pd(__mmask8 k, __m256d a, int imm);
VPERMILPD __m256d _mm256_maskz_permute_pd( __mmask8 k, __m256d a, int imm);
VPERMILPD __m128d _mm128_permute_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VPERMILPD __m128d _mm128_mask_permute_pd(__mmask8 k, __m128d a, int imm);
VPERMILPD __m128d _mm128_maskz_permute_pd( __mmask8 k, __m128d a, int imm);
VPERMILPD __m128d _mm_permute_pd (__m128d a, int control)
VPERMILPD __m256d _mm256_permute_pd(__m256d a, int control)
VPERMILPD __m128d _mm_permutevar_pd(__m128d a, __m128i control)
VPERMILPD __m256d _mm256_permutevar_pd(__m256d a, __m256i control);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4; additionally
#UD If VEX.W = 1.
EVEX-encoded instruction, see Exceptions Type E4NF.
#UD If either (E)VEX.vvvv != 1111B and with imm8.
### VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128</td>
<td>RVM V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 04 /r VPERMILPS xmm1, xmm2/m128, imm8</td>
<td>RM V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256</td>
<td>RVM V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 04 /r VPERMILPS ymm1, ymm2/m256, imm8</td>
<td>RM V/V</td>
<td>AVX</td>
<td>Permute single-precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV-RVM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV-RVM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 0C /r VPERMILPS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst</td>
<td>FV-RVM V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 04 /r VPERMILPS xmm1 (k1){z}, xmm2/m128/m32bcst, imm8</td>
<td>FV-RM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 04 /r VPERMILPS ymm1 (k1){z}, ymm2/m256/m32bcst, imm8</td>
<td>FV-RM V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 04 /r VPERMILPS zmm1 (k1){z}, zmm2/m512/m32bcst, imm8</td>
<td>FV-RM V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
**Description**

*(variable control version)*

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 5-26). Each control determines which of the source element in an input quadruple is selected for the destination element. Each quadruple of source elements must lie in the same 128-bit region as the destination.

**EVEX version:** The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

*(immediate control version)*

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

**VEX version:** The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

**EVEX version:** The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

**Note:** For the imm8 version, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.
Operation

Select4(SRC, control) {
CASE (control[1:0]) OF
  0: TMP ← SRC[31:0];
  1: TMP ← SRC[63:32];
  2: TMP ← SRC[95:64];
  3: TMP ← SRC[127:96];
ESAC;
RETURN TMP
}

VPERMILPS (EVEX immediate versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC1 *is memory*)
    THEN TMP_SRC1[i+31:i] ← SRC1[31:0];
    ELSE TMP_SRC1[i+31:i] ← SRC1[i+31:i];
  FI;
ENDFOR;

TMP_DEST[31:0] ← Select4(TMP_SRC1[127:0], imm8[1:0]);
TMP_DEST[95:64] ← Select4(TMP_SRC1[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC1[127:0], imm8[7:6]);
FI;

IF VL >= 256
  TMP_DEST[159:128] ← Select4(TMP_SRC1[255:128], imm8[1:0]);
FI;

IF VL >= 512
  TMP_DEST[287:256] ← Select4(TMP_SRC1[383:256], imm8[1:0]);
FI;

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[i] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*
        THEN *DEST[i+31:i] remains unchanged*
        ELSE DEST[i+31:i] ← 0 ;zeroing-masking
      FI;
  FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0
VPERMILPS (256-bit immediate version)
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[255:224] ← Select4(SRC1[255:128], imm8[7:6]);

VPERMILPS (128-bit immediate version)
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC1[127:0], imm8[7:6]);
DEST[MAX_VL-1:128] ← 0

VPERMILPS (EVEX variable versions)
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0];
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i];
  FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], TMP_SRC2[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], TMP_SRC2[33:32]);
TMP_DEST[95:64] ← Select4(SRC1[127:0], TMP_SRC2[65:64]);
TMP_DEST[127:96] ← Select4(SRC1[127:0], TMP_SRC2[97:96]);
IF VL >= 256
FI;
IF VL >= 512
  TMP_DEST[287:256] ← Select4(SRC1[383:256], TMP_SRC2[257:256]);
  TMP_DEST[319:288] ← Select4(SRC1[383:256], TMP_SRC2[289:288]);
  TMP_DEST[479:448] ← Select4(SRC1[511:384], TMP_SRC2[449:448]);
  TMP_DEST[511:480] ← Select4(SRC1[511:384], TMP_SRC2[481:480]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*
        THEN *DEST[i+31:i] remains unchanged*
      ELSE DEST[i+31:i] ← 0 ;zeroing-masking
      FI;
  FI;
VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

5-380 Vol. 2C

INSTRUCTION SET REFERENCE, V-Z

VPERMILPS (256-bit variable version)

DEST[31:0] <- Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] <- Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] <- Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] <- Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAX_VL-1:256] <- 0

VPERMILPS (128-bit variable version)

DEST[31:0] <- Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] <- Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] <- Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] <- Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAX_VL-1:128] <- 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPS __m512 __mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 __mm512_mask_permute_ps( __m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 __mm512_mask_permutevar_ps( __m512 i, __m512 a);
VPERMILPS __m512 __mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 __mm512_maskz_permutevar_ps( __mmask16 k, __m512 a);
VPERMILPS __m256 __mm256_permute_ps( __m256 s, __m512 a, int imm);
VPERMILPS __m256 __mm256_mask_permute_ps( __m512 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __mm256_mask_permutevar_ps( __m256 s, __m512 i, __m256 a);
VPERMILPS __m256 __mm256_maskz_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 __mm256_permutevar_ps( __m256 a, __m512 control);
VPERMILPS __m256 __mm256_permutevar_ps( __m256 a, __m256 control);
VPERMILPS __m128 __mm128_permute_ps( __m128 s, __m512 a, int imm);
VPERMILPS __m128 __mm128_mask_permute_ps( __m512 s, __mmask16 k, __m128 a, int imm);
VPERMILPS __m128 __mm128_maskz_permute_ps( __mmask16 k, __m128 a, int imm);
VPERMILPS __m128 __mm128_mask_permutevar_ps( __m128 s, __m512 i, __m128 a);
VPERMILPS __m128 __mm128_maskz_permutevar_ps( __mmask16 k, __m512 i, __m128 a);
VPERMILPS __m128 __mm128_permutevar_ps( __m128 a, __m512 control);
VPERMILPS __m128 __mm128_permutevar_ps( __m128 a, __m128 control);
VPERMILPS __m128 __mm128_permutevar_ps( __m128 a, __m128i control);
VPERMILPS __m128 __mm128_permutevar_ps( __m128 a, __m128i control);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;
#UD If VEX.W = 1.
EVEX-encoded instruction, see Exceptions Type E4NF.
#UD If either (E)VEX.vvvv ! = 1111B and with imm8.
VPERMPD—Permute Double-Precision Floating-Point Elements

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1, ymm2/m256, imm8</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permute double-precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>FV-RM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.S12.66.0F3A.W1 01 /r ib VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8</td>
<td>FV-RMI</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double-precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 16 /r VPERMPD ymm1 {k1}{z}, ymm3/m256/m64bcst</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm1 and store the result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 16 /r VPERMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double-precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RMI</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The imm8 version: Copies quadword elements of double-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAX_VL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Operation

**VPERMPD (EVEX - imm8 control forms)**

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN TMP_SRC[i+63:i] ← SRC[63:0];
    ELSE TMP_SRC[i+63:i] ← SRC[i+63:i];
  FI;
ENDFOR;

TMP_DEST[63:0] ← (TMP_SRC[256:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] ← (TMP_SRC[256:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[255:192] ← (TMP_SRC[256:0] >> (IMM8[7:6] * 64))[63:0];

IF VL >= 512
  TMP_DEST[319:256] ← (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
FI;

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0 ;zeroing-masking
      FI;
    FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0

**VPERMPD (EVEX - vector control forms)**

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[63:0] ← (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] ← (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
  TMP_DEST[191:128] ← (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
  TMP_DEST[255:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;

IF VL = 512
  TMP_DEST[63:0] ← (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
VPERMPD—Permute Double-Precision Floating-Point Elements

INSTRUCTION SET REFERENCE, V-Z

Vol. 2C 5-383

VPERMPD (VEX.256 encoded version)

DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAX_VL-1:256] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPD __m512d _mm512_permutex_pd(__m512d a, int imm);
VPERMPD __m512d _mm512_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_maskz_permutex_pd(__mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_permutexvar_pd(__m512i i, __m512d a);
VPERMPD __m512d _mm512_mask_permutexvar_pd(__m512d s, __mmask16 k, __m512i i, __m512d a);
VPERMPD __m512d _mm512_permutexvar_epi64(__m256d a, int imm);
VPERMPD __m512d _mm512_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256d a, int imm);
VPERMPD __m512d _mm512_maskz_permutexvar_epi64(__mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d _mm256_permutex_epi64(__m256i i, __m256d a);
VPERMPD __m256d _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d _mm256_maskz_permutexvar_epi64(__mmask8 k, __m256i i, __m256d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally
#UD  If VEX.L = 0.
    If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E4NF.
#UD  If encoded with EVEX.128.
    If VEX.vvvv != 1111B and with imm8.
VPERMPS—Permute Single-Precision Floating-Point Elements

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F38.W0 16 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permute single-precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 16 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point elements in ymm3/m256/m256bcst using indexes in ymm2 and store the result in ymm1 subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 16 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values in zmm3/m512/m512bcst using indexes in zmm2 and store the result in zmm1 subject to write mask k1.</td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

## Description

Copies doubleword elements of single-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

**VEX.256 versions:** The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAX_VL-1:256) of the corresponding destination register are zeroed.

**EVEX encoded version:** The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

## Operation

**VPERMPS (EVEX forms)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0];
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i];
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[31:0] ← (TMP_SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
  TMP_DEST[63:32] ← (TMP_SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
  TMP_DEST[95:64] ← (TMP_SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
  TMP_DEST[127:96] ← (TMP_SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
  TMP_DEST[159:128] ← (TMP_SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
  TMP_DEST[255:224] ← (TMP_SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
VPERMPS—Permute Single-Precision Floating-Point Elements

Ft;

IF VL = 512

    TMP_DEST[31:0] \leftarrow (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
    TMP_DEST[63:32] \leftarrow (TMP_SRC2[511:0] >> (SRC1[35:32] * 32))[31:0];
    TMP_DEST[95:64] \leftarrow (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
    TMP_DEST[127:96] \leftarrow (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
    TMP_DEST[159:128] \leftarrow (TMP_SRC2[511:0] >> (SRC1[131:128] * 32))[31:0];
    TMP_DEST[191:160] \leftarrow (TMP_SRC2[511:0] >> (SRC1[163:160] * 32))[31:0];
    TMP_DEST[223:192] \leftarrow (TMP_SRC2[511:0] >> (SRC1[195:192] * 32))[31:0];
    TMP_DEST[255:224] \leftarrow (TMP_SRC2[511:0] >> (SRC1[227:224] * 32))[31:0];
    TMP_DEST[287:256] \leftarrow (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
    TMP_DEST[319:288] \leftarrow (TMP_SRC2[511:0] >> (SRC1[291:288] * 32))[31:0];
    TMP_DEST[351:320] \leftarrow (TMP_SRC2[511:0] >> (SRC1[323:320] * 32))[31:0];
    TMP_DEST[383:352] \leftarrow (TMP_SRC2[511:0] >> (SRC1[355:352] * 32))[31:0];
    TMP_DEST[415:384] \leftarrow (TMP_SRC2[511:0] >> (SRC1[387:384] * 32))[31:0];
    TMP_DEST[447:416] \leftarrow (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
    TMP_DEST[479:448] \leftarrow (TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
    TMP_DEST[511:480] \leftarrow (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];

Fi;

FOR j \leftarrow 0 TO KL-1

    i \leftarrow j * 32

    IF k1[j] OR *no writemask*

        THEN DEST[i+31:i] \leftarrow TMP_DEST[i+31:i];

        ELSE

            IF *merging-masking* ; merging-masking

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

            ELSE ; zeroing-masking

                DEST[i+31:i] \leftarrow 0 ;zeroing-masking

            FI;

        FI;

    FI;

ENDFOR

DEST[MAX_VL-1:VL] \leftarrow 0

VPERMPS (VEX.256 encoded version)

DEST[31:0] \leftarrow (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] \leftarrow (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] \leftarrow (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] \leftarrow (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] \leftarrow (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] \leftarrow (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] \leftarrow (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] \leftarrow (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAX_VL-1:256] \leftarrow 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMPS __m512 _mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 _mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 _mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 _mm256_permutexvar_ps(__m256i i, __m256 a);
VPERMPS __m256 _mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMPS __m256 _mm256_maskz_permutexvar_ps(__mmask8 k, __m256 i, __m256 a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally
#UD If VEX.L = 0.
EVEX-encoded instruction, see Exceptions Type E4NF.
**VPERMQ—Qwords Element Permutation**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W1 00 / r ib</td>
<td>RMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permute qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3B.W1 00 / r ib</td>
<td>FV-RM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3B.W1 00 / r ib</td>
<td>FV-RMI</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute qwords in zmm2/m512/m64bcst using indexes in imm8 and store the result in zmm1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3B.W1 36 / r</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3B.W1 36 / r</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute qwords in zmm3/m512/m64bcst using indexes in zmm2 and store the result in zmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RMI</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RVM</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAX_VL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.
Operation

VPERMQ (EVEX - imm8 control forms)

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN TMP_SRC[i+63:i] ← SRC[i+63:i];
    ELSE TMP_SRC[i+63:i] ← SRC[i+63:i];
  FI;
ENDFOR;

  TMP_DEST[63:0] ← (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
  TMP_DEST[127:64] ← (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
  TMP_DEST[255:192] ← (TMP_SRC[255:0] >> (IMM8[7:6] * 64))[63:0];

IF VL >= 512
  TMP_DEST[319:256] ← (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
FI;

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE IF *merging-masking*
      THEN *DEST[i+63:i] remains unchanged*
    ELSE IF *merging-masking*
      THEN *DEST[i+63:i] remains unchanged*
    ELSE FI;
  FI;
ENDFOR

DEST[MAX_VL-1:VL] ← 0

VPERMQ (EVEX - vector control forms)

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] ← SRC2[i+63:i];
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[63:0] ← (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] ← (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
  TMP_DEST[191:128] ← (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
  TMP_DEST[255:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;

IF VL = 512
  TMP_DEST[63:0] ← (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
  TMP_DEST[127:64] ← (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
  TMP_DEST[191:128] ← (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
  TMP_DEST[255:192] ← (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
VPERMQ—Qwords Element Permutation

INSTRUCTION SET REFERENCE, V-Z

VPERMQ (VEX.256 encoded version)

DEST[63:0] ← (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ← (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[255:192] ← (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAX_VL-1:256] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMQ __m512i _mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i _mm512_mask_permutex_epi64( __m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_maskz_permutex_epi64( __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i _mm512_mask_permutexvar_epi64( __m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i _mm512_maskz_permutexvar_epi64( __mmask8 k, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally
#UD If VEX.L = 0.
If VEX.vvvv != 1111B.
EVEX-encoded instruction, see Exceptions Type E4NF.
#UD If encoded with EVEX.128.
If VEX.vvvv != 1111B and with imm8.
VPEXPANDD—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 99 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPEXPANDD xmm1 {k1}{z}, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 99 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>VPEXPANDD ymm1 {k1}{z}, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 99 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>VPEXPANDD zmm1 {k1}{z}, zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDD (EVEX encoded versions)

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

k ← 0

FOR j ← 0 TO KL-1

  i ← j * 32

  IF k1[j] OR *no writemask*

    THEN

      DEST[i+31:i] ← SRC[k+31:k];

      k ← k + 32

    ELSE

      IF *merging-masking* ; merging-masking

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

        ELSE ; zeroing-masking

          DEST[i+31:i] ← 0

      FI

  FI

ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDD __m512i _mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_maskz_expandloadu_epi32(__mmask16 k, void * a);
VPEXPANDD __m512i _mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VPEXPANDD __m512i _mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
VPEXPANDD __m256i _mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m256i _mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i _mm256_maskz_expand_epi32(__mmask8 k, __m256i a);
VPEXPANDD __m128i _mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i _mm_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m128i _mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VPEXPANDD __m128i _mm_maskz_expand_epi32(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.
VPEXPANDQ—Load Sparse Packed Quadword Integer Values from Dense Memory / Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 89 /r VPEXPANDQ xmm1 {k1}[z], xmm2/m128</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 89 /r VPEXPANDQ ymm1 {k1}[z], ymm2/m256</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 89 /r VPEXPANDQ zmm1 {k1}[z], zmm2/m512</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

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

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

**Operation**

**VPEXPANDQ (EVEX encoded versions)**

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

k ← 0

FOR j ← 0 TO KL-1

i ← j * 64

IF k1[j] OR *no writemask*

THEN

DEST[i+63:j] ← SRC[k+63:k];

k ← k + 64

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking 

THEN DEST[i+63:j] ← 0

FI

FI;

ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDQ __m512i _mm512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskexpand_epi64 (__m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m512i _mm512_maskz_expand_epi64 (__mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskz_expandloadu_epi64( __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_mask_expand_epi64(__m256i s, __mmask8 k, __m256i a);
VPEXPANDQ __m256i _mm256_maskz_expand_epi64( __mmask8 k, __m256i a);
VPEXPANDQ __m128i _mm_mask_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskz_expandloadu_epi64( __mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskexpand_epi64 (__m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm_maskz_expand_epi64( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD If EVEX.vvvv != 1111B.
### VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 44 /r VPLZCNTD xmm1 [k1][z], xmm2/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 44 /r VPLZCNTD ymm1 [k1][z], ymm2/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 44 /r VPLZCNTD zmm1 [k1][z], zmm2/m512/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 44 /r VPLZCNTQ xmm1 [k1][z], xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 44 /r VPLZCNTQ ymm1 [k1][z], ymm2/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 44 /r VPLZCNTQ zmm1 [k1][z], zmm2/m512/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

#### Description

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

- **EVEX.512 encoded version:** The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.
- **EVEX.256 encoded version:** The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.
- **EVEX.128 encoded version:** The source operand is an XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is an XMM register, conditionally updated using writemask k1.

**EVEX.vvvv** is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPLZCNTD
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
   i ← j*32
   IF MaskBit(j) OR *no writemask*
      THEN
         temp ← 32
         DEST[i+31:i] ← 0
         WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
            DO
               temp ← temp - 1
               DEST[i+31:i] ← DEST[i+31:i] + 1
            OD
      ELSE
         IF *merging-masking*
            THEN *DEST[i+31:i] remains unchanged*
            ELSE DEST[i+31:i] ← 0
         FI
      FI
   ENDFOR
DEST[MAX_VL-1:VL] ← 0

VPLZCNTQ
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
   i ← j*64
   IF MaskBit(j) OR *no writemask*
      THEN
         temp ← 64
         DEST[i+63:i] ← 0
         WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
            DO
               temp ← temp - 1
               DEST[i+63:i] ← DEST[i+63:i] + 1
            OD
      ELSE
         IF *merging-masking*
            THEN *DEST[i+63:i] remains unchanged*
            ELSE DEST[i+63:i] ← 0
         FI
      FI
   ENDFOR
DEST[MAX_VL-1:VL] ← 0
VPLZCNTD __m512i _mm512_lzcnt_epi32(__m512i a);
VPLZCNTD __m512i _mm512_mask_lzcnt_epi32(__m512i s, __mmask16 m, __m512i a);
VPLZCNTD __m512i _mm512_maskz_lzcnt_epi32(__mmask16 m, __m512i a);
VPLZCNTQ __m512i _mm512_lzcnt_epi64(__m512i a);
VPLZCNTQ __m512i _mm512_mask_lzcnt_epi64(__m512i s, __mmask8 m, __m512i a);
VPLZCNTQ __m512i _mm512_maskz_lzcnt_epi64(__mmask8 m, __m512i a);
VPLZCNTD __m256i _mm256_lzcnt_epi32(__m256i a);
VPLZCNTD __m256i _mm256_mask_lzcnt_epi32(__m256i s, __mmask8 m, __m256i a);
VPLZCNTD __m256i _mm256_maskz_lzcnt_epi32(__mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_lzcnt_epi64(__m256i a);
VPLZCNTQ __m256i _mm256_mask_lzcnt_epi64(__m256i s, __mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_maskz_lzcnt_epi64(__mmask8 m, __m256i a);
VPLZCNTD __m128i _mm_lzcnt_epi32(__m128i a);
VPLZCNTD __m128i _mm_mask_lzcnt_epi32(__m128i s, __mmask8 m, __m128i a);
VPLZCNTD __m128i _mm_maskz_lzcnt_epi32(__mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_lzcnt_epi64(__m128i a);
VPLZCNTQ __m128i _mm_mask_lzcnt_epi64(__m128i s, __mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_maskz_lzcnt_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.
### VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 8C /r VPMASKMOV xmm1, xmm2, m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load dword values from m128 using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 8C /r VPMASKMOVD ymm1, ymm2, m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load dword values from m256 using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 8C /r VPMASKMOVQ xmm1, xmm2, m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load qword values from m128 using mask in xmm2 and store in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 8C /r VPMASKMOVDQ ymm1, ymm2, m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load qword values from m256 using mask in ymm2 and store in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W0 8E /r VPMASKMOVD m128, xmm1, xmm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally store dword values from xmm2 using mask in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 8E /r VPMASKMOVDQ m256, ymm1, ymm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally store qword values from ymm2 using mask in ymm1.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 8E /r VPMASKMOVQ m128, xmm1, xmm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally store qword values from xmm2 using mask in xmm1.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 8E /r VPMASKMOVDQQ m256, ymm1, ymm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally store qword values from ymm2 using mask in ymm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>MVR</td>
<td>ModRM:r/m (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are either XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.
VMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

**Operation**

**VPMASKMOVD - 256-bit load**

DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0

**VPMASKMOVD - 128-bit load**

DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[VLMAX-1:128] ← 0

**VPMASKMOVQ - 256-bit load**

DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0

**VPMASKMOVQ - 128-bit load**

DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[VLMAX-1:128] ← 0

**VPMASKMOVD - 256-bit store**

IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ← SRC2[159:128]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
VPMASKMOVD - 128-bit store
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

VPMASKMOVD - 256-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

VPMASKMOVQ - 128-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ← SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent
VPMASKMOVD: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)
VPMASKMOVD: void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)
VPMASKMOVQ: __m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);
VPMASKMOVQ: void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type 6 (No AC# reported for any mask bit combinations).
VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1</td>
<td>RM/V/V</td>
<td>AVX512VL</td>
<td>Sets each byte in XMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1</td>
<td>RM/V/V</td>
<td>AVX512VL</td>
<td>Sets each byte in YMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1</td>
<td>RM/V/V</td>
<td>AVX512BW</td>
<td>Sets each byte in ZMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1</td>
<td>RM/V/V</td>
<td>AVX512VL</td>
<td>Sets each word in XMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1</td>
<td>RM/V/V</td>
<td>AVX512VL</td>
<td>Sets each word in YMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 28 /r VPMOVM2W zmm1, k1</td>
<td>RM/V/V</td>
<td>AVX512BW</td>
<td>Sets each word in ZMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1</td>
<td>RM/V/V</td>
<td>AVX512VL</td>
<td>Sets each doubleword in XMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1</td>
<td>RM/V/V</td>
<td>AVX512VL</td>
<td>Sets each doubleword in YMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1</td>
<td>RM/V/V</td>
<td>AVX512DQ</td>
<td>Sets each doubleword in ZMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1</td>
<td>RM/V/V</td>
<td>AVX512VL</td>
<td>Sets each quadword in XMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1</td>
<td>RM/V/V</td>
<td>AVX512VL</td>
<td>Sets each quadword in YMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 38 /r VPMOVM2Q zmm1, k1</td>
<td>RM/V/V</td>
<td>AVX512DQ</td>
<td>Sets each quadword in ZMM1 to all 1’s or all 0’s based on the value of the corresponding bit in k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1’s or all 0’s depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register.

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

VPMOVM2B (EVEX encoded versions)

\[(K_L, V_L) = (16, 128), (32, 256), (64, 512)\]

FOR \( j \leftarrow 0 \) TO \( K_L - 1 \)
\[ i \leftarrow j \times 8 \]
IF \( SRC[j] \)
THEN \( \text{DEST}[i+7:i] \leftarrow -1 \)
ELSE \( \text{DEST}[i+7:i] \leftarrow 0 \)
FI;
ENDFOR

\( \text{DEST}[\text{MAX}_V_L:V_L] \leftarrow 0 \)

VPMOVM2W (EVEX encoded versions)

\[(K_L, V_L) = (8, 128), (16, 256), (32, 512)\]

FOR \( j \leftarrow 0 \) TO \( K_L - 1 \)
\[ i \leftarrow j \times 16 \]
IF \( SRC[j] \)
THEN \( \text{DEST}[i+15:i] \leftarrow -1 \)
ELSE \( \text{DEST}[i+15:i] \leftarrow 0 \)
FI;
ENDFOR

\( \text{DEST}[\text{MAX}_V_L:V_L] \leftarrow 0 \)

VPMOVM2D (EVEX encoded versions)

\[(K_L, V_L) = (4, 128), (8, 256), (16, 512)\]

FOR \( j \leftarrow 0 \) TO \( K_L - 1 \)
\[ i \leftarrow j \times 32 \]
IF \( SRC[j] \)
THEN \( \text{DEST}[i+31:i] \leftarrow -1 \)
ELSE \( \text{DEST}[i+31:i] \leftarrow 0 \)
FI;
ENDFOR

\( \text{DEST}[\text{MAX}_V_L:V_L] \leftarrow 0 \)

VPMOVM2Q (EVEX encoded versions)

\[(K_L, V_L) = (2, 128), (4, 256), (8, 512)\]

FOR \( j \leftarrow 0 \) TO \( K_L - 1 \)
\[ i \leftarrow j \times 64 \]
IF \( SRC[j] \)
THEN \( \text{DEST}[i+63:i] \leftarrow -1 \)
ELSE \( \text{DEST}[i+63:i] \leftarrow 0 \)
FI;
ENDFOR

\( \text{DEST}[\text{MAX}_V_L:V_L] \leftarrow 0 \)
**Intel C/C++ Compiler Intrinsic Equivalents**

VPMOVM2B __m512i __mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i __mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i __mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i __mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i __mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i __mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i __mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i __mm256_movm_epi16(__mmask16);
VPMOVM2B __m128i __mm_movm_epi8(__mmask16);
VPMOVM2D __m128i __mm_movm_epi32(__mmask8);
VPMOVM2Q __m128i __mm_movm_epi64(__mmask8);
VPMOVM2W __m128i __mm_movm_epi16(__mmask8);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E7NM

#UD If EVEX.vvvv != 1111B.
VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 29 /r VPMOVB2M k1, xmm1</td>
<td>RM/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 29 /r VPMOVB2M k1, ymm1</td>
<td>RM/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 29 /r VPMOVB2M k1, zmm1</td>
<td>RM/V</td>
<td>AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 29 /r VPMOVW2M k1, xmm1</td>
<td>RM/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 29 /r VPMOVW2M k1, ymm1</td>
<td>RM/V</td>
<td>AVX512VL, AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 29 /r VPMOVW2M k1, zmm1</td>
<td>RM/V</td>
<td>AVX512BW</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 39 /r VPMOVD2M k1, xmm1</td>
<td>RM/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 39 /r VPMOVD2M k1, ymm1</td>
<td>RM/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 39 /r VPMOVD2M k1, zmm1</td>
<td>RM/V</td>
<td>AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 39 /r VPMOVQ2M k1, xmm1</td>
<td>RM/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 39 /r VPMOVQ2M k1, ymm1</td>
<td>RM/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 39 /r VPMOVQ2M k1, zmm1</td>
<td>RM/V</td>
<td>AVX512DQ</td>
<td>Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register. EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVB2M (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  IF SRC[i+7]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPMOVW2M (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF SRC[i+15]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPMOVD2M (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF SRC[i+31]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPMOVQ2M (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF SRC[i+63]
    THEN DEST[j] ← 1
    ELSE DEST[j] ← 0
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalents

VPMPOVB2M __mmask64 _mm512_movepi8_mask(__m512i);
VPMPOVD2M __mmask16 _mm512_movepi32_mask(__m512i);
VPMPOVQ2M __mmask8 _mm512_movepi64_mask(__m512i);
VPMPOVw2M __mmask32 _mm512_movepi16_mask(__m512i);
VPMPOVB2M __mmask32 _mm256_movepi8_mask(__m256i);
VPMPOVD2M __mmask8 _mm256_movepi32_mask(__m256i);
VPMPOVQ2M __mmask8 _mm256_movepi64_mask(__m256i);
VPMPOVw2M __mmask16 _mm256_movepi16_mask(__m256i);
VPMPOVB2M __mmask16 _mmMovepi8_mask(__m128i);
VPMPOVD2M __mmask8 _mm_movepi32_mask(__m128i);
VPMPOVQ2M __mmask8 _mm_movepi64_mask(__m128i);
VPMPOVw2M __mmask8 _mm_movepi16_mask(__m128i);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E7NM

#UD          If EVEX.vvv != 111B.
**VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 32 /r VPMOVQB xmm1/m16 {k1}{z}, xmm2</td>
<td>OVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed quad-word integers from xmm2 into 2 packed byte integers in xmm1/m16 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 22 /r VPMOVSQB xmm1/m16 {k1}{z}, xmm2</td>
<td>OVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed byte integers in xmm1/m16 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m16 {k1}{z}, xmm2</td>
<td>OVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned byte integers in xmm1/m16 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 32 /r VPMOVQB xmm1/m32 {k1}{z}, ymm2</td>
<td>OVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed quad-word integers from ymm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 22 /r VPMOVSQB xmm1/m32 {k1}{z}, ymm2</td>
<td>OVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m32 {k1}{z}, ymm2</td>
<td>OVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 32 /r VPMOVQB xmm1/m64 {k1}{z}, zmm2</td>
<td>OVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed quad-word integers from zmm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 22 /r VPMOVSQB xmm1/m64 {k1}{z}, zmm2</td>
<td>OVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m64 {k1}{z}, zmm2</td>
<td>OVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>OVM</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX_VL-1:64) of the destination are zeroed.

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

VPMOVQB instruction (EVEX encoded versions) when dest is a register

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

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\(i \leftarrow j * 8\)

\(m \leftarrow j * 64\)

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

THEN \(DEST[i+7:i] \leftarrow \text{TruncateQuadWordToByte} (SRC[m+63:m])\)

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+7:i] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\(DEST[i+7:i] \leftarrow 0\)

FI

ENDFOR

DEST[MAX_VL-1:VL/8] \leftarrow 0;

VPMOVQB instruction (EVEX encoded versions) when dest is memory

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

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\(i \leftarrow j * 8\)

\(m \leftarrow j * 64\)

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

THEN \(DEST[i+7:i] \leftarrow \text{TruncateQuadWordToByte} (SRC[m+63:m])\)

ELSE

*DEST[i+7:i] remains unchanged* ; merging-masking

ENDFOR

VPMOVVSQB instruction (EVEX encoded versions) when dest is a register

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

FOR \(j \leftarrow 0 \) TO \(KL-1\)

\(i \leftarrow j * 8\)

\(m \leftarrow j * 64\)

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

THEN \(DEST[i+7:i] \leftarrow \text{SaturateSignedQuadWordToByte} (SRC[m+63:m])\)

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+7:i] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\(DEST[i+7:i] \leftarrow 0\)

FI

ENDFOR

DEST[MAX_VL-1:VL/8] \leftarrow 0;
VPMOVQSQB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVUSQB instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] ← 0
      FI
  FI;
ENDFOR

DEST[MAX_VL-1:VL/8] ← 0;

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQB __m128i _mm512_cvtepi64_epi8( __m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8( __m512i s, __mmask8 k, __m512i a);
VPMOVQB void _mm512_maskz_cvtepi64_epi8( void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8( __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8( void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8( __m512i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8( __mmask8 k, __m512i a);

VPMOVQB __m128i _mm512_mask_cvtepi64_epi8( __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_epi8( void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8( __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8( __m512i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);

VPMOVSQB __m128i _mm512_cvtsepi64_epi8( __m512i a);
VPMOVSQB __m128i _mm512_mask_cvtsepi64_epi8( __m128i s, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtsepi64_epi8( __mmask8 k, __m512i a);
VPMOVSQB void _mm512_mask_cvtsepi64_storeu_epi8( void * d, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtsepi64_epi8( __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_mask_cvtsepi64_epi8( __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtsepi64_epi8( __mmask8 k, __m512i a);

VPMOVSQB __m128i _mm512_mask_cvtusepi64_epi8( __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtusepi64_epi8( __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_mask_cvtusepi64_epi8( __m512i s, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtusepi64_epi8( __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_mask_cvtusepi64_epi8( __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtusepi64_epi8( __mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>QVM</td>
<td>ModRMs/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW converts unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAX_VL-1:128/64/32) of the register destination are zeroed.

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

**VPMOVQw instruction (EVEX encoded versions) when dest is a register**

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

FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR

DEST[MAX_VL-1:VL/4] ← 0;

**VPMOVQw instruction (EVEX encoded versions) when dest is memory**

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

FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

**VPMOVQw instruction (EVEX encoded versions) when dest is a register**

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

FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR

DEST[MAX_VL-1:VL/4] ← 0;
VPMOVQW instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VPMOVUSQW instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] ← 0
      FI
    FI
ENDFOR
DEST[MAX_VL-1:VL/4] ← 0;

VPMOVUSQW instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQW __m128i __m512_cvtepi64_epi16(__m512i a);
VPMOVQW __m128i __m512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i __m512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void __m512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVQW __m128i __m512_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i __m512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void __m512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);

VPMOVSQW __m128i __m512_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i __m512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i __m512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW void __m512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i __m512_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i __m512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW void __m512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);

VPMOVUSQW __m128i __m512_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i __m512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i __m512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVUSQW void __m512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i __m512_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i __m512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVUSQW void __m512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);

VPMOVQD __m128i __m256_cvtusepi64_epi32(__m256i a);
VPMOVQD __m128i __m256_mask_cvtusepi64_epi32(__m256i a, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __m256_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __m256_mask_cvtusepi64_epi32(__m256i a, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __m256_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __m256_mask_cvtusepi64_epi32(__m256i a, __mmask8 k, __m256i b);
VPMOVQD __m128i __m256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __m256_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 35 /r</td>
<td>r</td>
<td></td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed quad-word integers from xmm2 into 2 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 25 /r</td>
<td>r</td>
<td></td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed double-word integers in xmm1/m64 using signed saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 15 /r</td>
<td>r</td>
<td></td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned double-word integers in xmm1/m64 using unsinged saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 35 /r</td>
<td>r</td>
<td></td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed quad-word integers from ymm2 into 4 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 25 /r</td>
<td>r</td>
<td></td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed double-word integers in xmm1/m128 using signed saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 15 /r</td>
<td>r</td>
<td></td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned double-word integers in xmm1/m128 using unsinged saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 35 /r</td>
<td>r</td>
<td></td>
<td>AVX512F</td>
<td>Converts 8 packed quad-word integers from zmm2 into 8 packed double-word integers in ymm1/m256 with truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 25 /r</td>
<td>r</td>
<td></td>
<td>AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed double-word integers in ymm1/m256 using signed saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 15 /r</td>
<td>r</td>
<td></td>
<td>AVX512F</td>
<td>Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned double-word integers in ymm1/m256 using unsinged saturation subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HVM</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAX_VL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
**Operation**

**VPMOVQD instruction (EVEX encoded version) reg-reg form**

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

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1
\]

\[
i \leftarrow j * 32
\]

\[
m \leftarrow j * 64
\]

\[
\text{IF } k1[j] \text{ OR } \text{*no writemask*}
\]

\[
\text{THEN } \text{DEST}[i+31:i] \leftarrow \text{TruncateQuadWordToDWord} (\text{SRC}[m+63:m])
\]

\[
\text{ELSE } \text{DEST}[i+31:i] \leftarrow 0
\]

\[
\text{FI}
\]

\[
\text{FI;}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAX}_V L-1:V L/2] \leftarrow 0;
\]

**VPMOVQD instruction (EVEX encoded version) memory form**

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

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1
\]

\[
i \leftarrow j * 32
\]

\[
m \leftarrow j * 64
\]

\[
\text{IF } k1[j] \text{ OR } \text{*no writemask*}
\]

\[
\text{THEN } \text{DEST}[i+31:i] \leftarrow \text{TruncateQuadWordToDWord} (\text{SRC}[m+63:m])
\]

\[
\text{ELSE } \text{DEST}[i+31:i] \leftarrow 0
\]

\[
\text{FI;}
\]

**VPMOVUSQD instruction (EVEX encoded version) reg-reg form**

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

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1
\]

\[
i \leftarrow j * 32
\]

\[
m \leftarrow j * 64
\]

\[
\text{IF } k1[j] \text{ OR } \text{*no writemask*}
\]

\[
\text{THEN } \text{DEST}[i+31:i] \leftarrow \text{SaturateSignedQuadWordToDWord} (\text{SRC}[m+63:m])
\]

\[
\text{ELSE}
\]

\[
\text{IF } \text{*merging-masking*}
\]

\[
\text{THEN } \text{DEST}[i+31:i] \leftarrow 0
\]

\[
\text{ELSE } \text{DEST}[i+31:i] \leftarrow 0
\]

\[
\text{FI}
\]

\[
\text{FI;}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAX}_V L-1:V L/2] \leftarrow 0;
\]

**VPMOVUSQD instruction (EVEX encoded version) memory form**

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

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1
\]

\[
i \leftarrow j * 32
\]

\[
m \leftarrow j * 64
\]

\[
\text{IF } k1[j] \text{ OR } \text{*no writemask*}
\]

\[
\text{THEN } \text{DEST}[i+31:i] \leftarrow \text{SaturateSignedQuadWordToDWord} (\text{SRC}[m+63:m])
\]

\[
\text{ELSE}
\]

\[
\text{IF } \text{*merging-masking*}
\]

\[
\text{THEN } \text{DEST}[i+31:i] \leftarrow 0
\]

\[
\text{ELSE } \text{DEST}[i+31:i] \leftarrow 0
\]

\[
\text{FI}
\]

\[
\text{FI;}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAX}_V L-1:V L/2] \leftarrow 0;
\]
VPMOVQD instruction (EVEX encoded version) memory form
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVUSQD instruction (EVEX encoded version) reg-reg form
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

VPMOVUSQD instruction (EVEX encoded version) memory form
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  m ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
**Intel C/C++ Compiler Intrinsic Equivalents**

VPMOVQD __m256i _mm512_cvtepi64_epi32( __m512i a);
VPMOVQD __m256i _mm512_mask_cvtepi64_epi32( __m256i s, __mmask8 k, __m512i a);
VPMOVQD __m256i _mm512_maskz_cvtepi64_epi32( __mmask8 k, __m512i a);
VPMOVQD void _mm512_mask_cvtepi64_storeu_epi32( void * d, __mmask8 k, __m512i a);
VPMOVQD __m256i _mm512_cvtepi64_epi32( __m512i a);
VPMOVQD __m256i _mm512_maskz_cvtepi64_epi32( __mmask8 k, __m512i a);
VPMOVQD __m256i _mm512_mask_cvtepi64_epi32( __m512i a);
VPMOVQD __m256i _mm512_maskz_cvtepi64_epi32( __mmask8 k, __m512i a);
VPMOVQD __m256i _mm512_maskz_cvtepi64_epi32( __mmask8 k, __m512i a);
VPMOVQD __m512i _mm256_cvtepi64_epi32( __m512i a);
VPMOVQD __m512i _mm_mask_cvtepi64_epi32( __m512i a, __mmask8 k, __m512i b);
VPMOVQD __m512i _mm_maskz_cvtepi64_epi32( __mmask8 k, __m512i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32( void * d, __mmask8 k, __m512i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
# VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 31 / r</td>
<td>QVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed double-word integers from xmm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVDB xmm1/m32 {k1}{z}, xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 21 / r</td>
<td>QVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed double-word integers from xmm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVSDB xmm1/m32 {k1}{z}, xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 11 / r</td>
<td>QVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned byte integers in xmm1/m32 using unsign saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVUSDB xmm1/m32 {k1}{z}, xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 31 / r</td>
<td>QVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed double-word integers from ymm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVDB ymm1/m64 {k1}{z}, ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 21 / r</td>
<td>QVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed signed double-word integers from ymm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVSDB ymm1/m64 {k1}{z}, ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 11 / r</td>
<td>QVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned byte integers in xmm1/m64 using unsign saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVUSDB ymm1/m64 {k1}{z}, ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 31 / r</td>
<td>QVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed double-word integers from zmm2 into 16 packed byte integers in xmm1/m128 with truncation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVDB zmm1/m128 {k1}{z}, zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 21 / r</td>
<td>QVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed signed double-word integers from zmm2 into 16 packed signed byte integers in xmm1/m128 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVSDB zmm1/m128 {k1}{z}, zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 11 / r</td>
<td>QVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned byte integers in xmm1/m128 using unsign saturation under writemask k1.</td>
</tr>
<tr>
<td>VPMOVUSDB zmm1/m128 {k1}{z}, zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>QVM</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

VPMOVDB down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMOVSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVUSDB converts unsigned double-word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX_VL-1:128/64/32) of the register destination are zeroed.

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

VPMOVDB instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j \leftarrow 0 \text{ TO } KL-1
  i \leftarrow j * 8
  m \leftarrow j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] \leftarrow \text{TruncateDoubleWordToByte} (SRC[m+31:m])
    ELSE
      IF *merging-mask* ; merging-mask
        THEN *DEST[i+7:i] remains unchanged* 
      ELSE *zeroing-mask* ; zeroing-mask
        DEST[i+7:i] \leftarrow 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] \leftarrow 0;

VPMOVDB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j \leftarrow 0 \text{ TO } KL-1
  i \leftarrow j * 8
  m \leftarrow j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] \leftarrow \text{TruncateDoubleWordToByte} (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-mask
  FI;
ENDFOR

VPMOVSDB instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j \leftarrow 0 \text{ TO } KL-1
  i \leftarrow j * 8
  m \leftarrow j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] \leftarrow \text{SaturateSignedDoubleWordToByte} (SRC[m+31:m])
    ELSE
      IF *merging-mask* ; merging-mask
        THEN *DEST[i+7:i] remains unchanged* 
      ELSE *zeroing-mask* ; zeroing-mask
        DEST[i+7:i] \leftarrow 0
      FI
  FI;
ENDFOR
DEST[MAX_VL-1:VL/4] \leftarrow 0;
VPMOVDB instruction (EVEX encoded versions) when dest is memory

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

FOR \(j \leftarrow 0\) TO \(KL-1\)
\[
i \leftarrow j \times 8
\]
\[
m \leftarrow j \times 32
\]
IF \(k1[j] \text{ OR *no writemask*}
\[
\text{THEN } \text{DEST}[i+7:i] \leftarrow \text{SaturateSignedDoubleWordToByte} \ (\text{SRC}[m+31:m])
\]
ELSE \(*\text{DEST}[i+7:i] \text{ remains unchanged* ; merging-masking}
\]
ENDFOR

VPMOVVSDB instruction (EVEX encoded versions) when dest is a register

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

FOR \(j \leftarrow 0\) TO \(KL-1\)
\[
i \leftarrow j \times 8
\]
\[
m \leftarrow j \times 32
\]
IF \(k1[j] \text{ OR *no writemask*}
\[
\text{THEN } \text{DEST}[i+7:i] \leftarrow \text{SaturateUnsignedDoubleWordToByte} \ (\text{SRC}[m+31:m])
\]
ELSE
\[
\text{IF *merging-masking* ; merging-masking}
\]
\[
\text{ELSE *zeroing-masking* ; zeroing-masking}
\]
\[
\text{DEST}[i+7:i] \leftarrow 0
\]
ENDFOR

ENDFOR

DEST[\(\text{MAX\_VL-1:VL/4}\) \leftarrow 0;

VPMOVUSDB instruction (EVEX encoded versions) when dest is memory

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

FOR \(j \leftarrow 0\) TO \(KL-1\)
\[
i \leftarrow j \times 8
\]
\[
m \leftarrow j \times 32
\]
IF \(k1[j] \text{ OR *no writemask*}
\[
\text{THEN } \text{DEST}[i+7:i] \leftarrow \text{SaturateUnsignedDoubleWordToByte} \ (\text{SRC}[m+31:m])
\]
ELSE \(*\text{DEST}[i+7:i] \text{ remains unchanged* ; merging-masking}
\]
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDB __m128i _mm512_cvtepi32_epi8( __m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8( __m128i s, __mmask16 k, __m512i a);
VPMOVDB _mm512_mask_cvtepi32_epi8( void * d, __mmask16 k, __m512i a);
VPMOVSB __m128i _mm512_cvtsepi32_epi8( __m512i a);
VPMOVSB __m128i _mm512_mask_cvtsepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVSB _mm512_mask_cvtsepi32_epi8( void * d, __mmask16 k, __m512i a);
VPMOVSB __m128i _mm512_maskz_cvtsepi32_epi8( __mmask16 k, __m512i a);
VPMOVSB __m128i _mm512_maskz_cvtsepi32_epi8( __mmask16 k, __m512i a);
VPMOVSB _mm512_mask_z_cvtsepi32_epi8( __mmask16 k, __m512i a);
VPMOVSB _mm512_cvtusepi32_epi8( __m512i a);
VPMOVSB __m128i _mm512_mask_cvtusepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVSB _mm512_mask_cvtusepi32_epi8( void * d, __mmask16 k, __m512i a);
VPMOVSB __m128i _mm512_maskz_cvtusepi32_epi8( __mmask16 k, __m512i a);
VPMOVSB _mm512_mask_cvtusepi32_epi8( void * d, __mmask16 k, __m512i a);
VPMOVSB __m128i _mm512_cvtepi32_epi8( __m512i a);
VPMOVSB __m128i _mm512_mask_cvtusepi32_epi8( __m512i a);
VPMOVSB __m128i _mm512_maskz_cvtepi32_epi8( __mmask16 k, __m512i a);
VPMOVSB _mm512_mask_cvtusepi32_epi8( void * d, __mmask16 k, __m512i a);
VPMOVSB __m128i _mm512_cvtepi32_epi8( __m512i a);
VPMOVSB _mm512_mask_cvtepi32_epi8( __m512i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 33 /r VPMOVDW xmm1/m64 [k1][z], xmm2</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed double-word integers from xmm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 23 /r VPMOVSDW xmm1/m64 [k1][z], xmm2</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed double-word integers from xmm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m64 [k1][z], xmm2</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 33 /r VPMOVDW xmm1/m128 [k1][z], ymm2</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed double-word integers from ymm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 23 /r VPMOVSDW xmm1/m128 [k1][z], ymm2</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed signed double-word integers from ymm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 13 /r VPMOVUSDW xmm1/m128 [k1][z], ymm2</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 33 /r VPMOVDW ymm1/m256 [k1][z], zmm2</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed double-word integers from zmm2 into 16 packed word integers in ymm1/m256 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 23 /r VPMOVSDW ymm1/m256 [k1][z], zmm2</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed signed double-word integers from zmm2 into 16 packed signed word integers in ymm1/m256 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW ymm1/m256 [k1][z], zmm2</td>
<td>HVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned word integers in ymm1/m256 using unsigned saturation under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HVM</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAX_VL-1:256/128/64) of the register destination are zeroed.

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

**VPMOVDW instruction (EVEX encoded versions) when dest is a register**

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

FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged* 
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR

DEST[MAX_VL-1:VL/2] ← 0;

**VPMOVDW instruction (EVEX encoded versions) when dest is memory**

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

FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

**VPMOVUSDW instruction (EVEX encoded versions) when dest is a register**

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

FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged* 
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR

DEST[MAX_VL-1:VL/2] ← 0;
VPMOVSDw instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            *DEST[i+15:i] remains unchanged* ; merging-masking
        FI;
    ENDFOR

VPMOVUSDw instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+15:i] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL/2] ← 0;

VPMOVUSDw instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            *DEST[i+15:i] remains unchanged* ; merging-masking
        FI;
    ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDW __m256i _mm512_cvtepi32_epi16(__m512i a);
VPMOVDW __m256i _mm512_mask_cvtepi32_epi16(__m512i s, __mmask16 k, __m512i a);
VPMOVDW void _mm512_maskz_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_cvtepi32_epi16(__m512i a);
VPMOVUSDW __m256i _mm512_mask_cvtepi32_epi16(__m512i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void _mm512_maskz_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m128i _mm256_cvtepi32_epi16(__m256i a);
VPMOVUSDW __m128i _mm256_mask_cvtepi32_epi16(__m256i a, __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm256_maskz_cvtepi32_epi16(__mmask8 k, __m256i b);
VPMOVUSDW void _mm256_maskz_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm_cvtepi32_epi16(__m128i a);
VPMOVUSDW __m128i _mm_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDW __m128i _mm_maskz_cvtepi32_epi16(__mmask8 k, __m128i b);
VPMOVUSDW void _mm_maskz_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.
VPMOV WB/VPMOVSwB/VPMOVUSWB—Down Convert Word to Byte

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 30 / r VPMOVWB xmm1/m64 [k1][z], xmm2</td>
<td>HVM V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Converts 8 packed word integers from xmm2 into 8 packed bytes in xmm1/m64 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 20 / r VPMOVSwB xmm1/m64 [k1][z], xmm2</td>
<td>HVM V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Converts 8 packed signed word integers from xmm2 into 8 packed signed bytes in xmm1/m64 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 10 / r VPMOVUSWB xmm1/m64 [k1][z], xmm2</td>
<td>HVM V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Converts 8 packed unsigned word integers from xmm2 into 8 packed unsigned bytes in 8mm1/m64 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 30 / r VPMOVWB xmm1/m128 [k1][z], ymm2</td>
<td>HVM V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Converts 16 packed word integers from ymm2 into 16 packed bytes in xmm1/m128 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 20 / r VPMOVSwB xmm1/m128 [k1][z], ymm2</td>
<td>HVM V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Converts 16 packed signed word integers from ymm2 into 16 packed signed bytes in xmm1/m128 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 10 / r VPMOVUSWB xmm1/m128 [k1][z], ymm2</td>
<td>HVM V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Converts 16 packed unsigned word integers from ymm2 into 16 packed unsigned bytes in xmm1/m128 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 30 / r VPMOVWB ymm1/m256 [k1][z], zmm2</td>
<td>HVM V/V</td>
<td>AVX512Bw</td>
<td>Converts 32 packed word integers from zmm2 into 32 packed bytes in ymm1/m256 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 20 / r VPMOVSwB ymm1/m256 [k1][z], zmm2</td>
<td>HVM V/V</td>
<td>AVX512Bw</td>
<td>Converts 32 packed signed word integers from zmm2 into 32 packed signed bytes in ymm1/m256 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 10 / r VPMOVUSWB ymm1/m256 [k1][z], zmm2</td>
<td>HVM V/V</td>
<td>AVX512Bw</td>
<td>Converts 32 packed unsigned word integers from zmm2 into 32 packed unsigned bytes in ymm1/m256 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HVM</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VPMOV WB down converts 16-bit integers into packed bytes using truncation. VPMOVSwB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAX_VL-1:256/128/64) of the register destination are zeroed.

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

**VPMOVWB instruction (EVEX encoded versions) when dest is a register**

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

FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask* 
     THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
     ELSE
       IF *merging-masking* ; merging-masking
          THEN *DEST[i+7:i] remains unchanged* 
          ELSE *zeroing-masking* ; zeroing-masking
             DEST[i+7:i] = 0
       FI
  FI;
ENDFOR

DEST[MAX_VL-1:VL/2] ← 0;

**VPMOVWB instruction (EVEX encoded versions) when dest is memory**

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

FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask* 
     THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
     ELSE
       *DEST[i+7:i] remains unchanged* ; merging-masking
     FI;
ENDFOR

**VPMOVWSB instruction (EVEX encoded versions) when dest is a register**

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

FOR j ← 0 TO KL-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask* 
     THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
     ELSE 
       IF *merging-masking* ; merging-masking 
          THEN *DEST[i+7:i] remains unchanged* 
          ELSE *zeroing-masking* ; zeroing-masking
             DEST[i+7:i] = 0
       FI
  FI;
ENDFOR

DEST[MAX_VL-1:VL/2] ← 0;
VPMOVSwB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO Kl-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVUSwB instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO Kl-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] = 0
      FI
  FI;
ENDFOR
 DEST[MAX_VL-1:VL/2] ← 0;

VPMOVUSwB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO Kl-1
  i ← j * 8
  m ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVUSB __m256i _mm512_cvtepi16_epi8(__m512i a);
VPMOVUSB __m256i _mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSB __m256i _mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSB __m256i _mm512_maskz_cvtusepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSB __m128i _mm256_cvtusepi16_epi8(__m256i a);
VPMOVUSB __m128i _mm256_mask_cvtusepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSB __m128i _mm256_maskz_cvtusepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSB __m128i _mm256_mask_cvtsepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSB __m128i _mm256_maskz_cvtsepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSB __m128i _mm256_cvtepi16_epi8(__m256i a);
VPMOVUSB __m128i _mm256_mask_cvtepi16_epi8(__m256i a, __mmask8 k, __m256i b);
VPMOVUSB __m128i _mm256_maskz_cvtepi16_epi8(__mmask8 k, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF

#UD If EVEX.vvvv != 1111B.
### INSTRUCTION SET REFERENCE, V-Z

#### PROLD/PROLVD/PROLQ/PROLVQ—Bit Rotate Left

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 15 /r</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDD.128.66.0F.W0 72 /1 ib</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 15 /r</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDD.128.66.0F.W1 72 /1 ib</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 15 /r</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDD.256.66.0F.W0 72 /1 ib</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 15 /r</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDD.256.66.0F.W1 72 /1 ib</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 15 /r</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDD.512.66.0F.W0 72 /1 ib</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate left of doublewords in zmm2/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 15 /r</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDD.512.66.0F.W1 72 /1 ib</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV-VMI</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (R)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.
EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:128) of the corresponding ZMM register are zeroed.
EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.
EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC << COUNT) | (SRC >> (32 - COUNT));

LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC << COUNT) | (SRC >> (64 - COUNT));

VPROLD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[31:0], imm8)
      ELSE DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
    FI;
    ELSE
      IF *merging-masking*; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*; zeroing-masking
          DEST[i+31:i] ← 0
        FI;
      FI;
  ENDFOR
DEST[MAX_VL-1:VL] ← 0
VPROLVD (EVEX encoded versions)

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

FOR \(j \leftarrow 0\) TO \(KL-1\)
  \(i \leftarrow j \times 32\)
  IF \(k1[j] \text{ OR *no writemask* THEN}\)
    IF (\(EVEX.b = 1\) AND (\(SRC2 \text{ *is memory*})
      \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
      THEN DEST\([i+31:i]\) \leftarrow \text{LEFT Rotate Dwords}(SRC1\([i+31:i]\), SRC2\([31:0]\))
      ELSE \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
      \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
      \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST\([i+31:i]\] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
        \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
        \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
      FI
  FI;
ENDFOR

DEST\([\text{MAX}_V L-1:VL]\) \leftarrow 0

VPROLOQ (EVEX encoded versions)

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

FOR \(j \leftarrow 0\) TO \(KL-1\)
  \(i \leftarrow j \times 64\)
  IF \(k1[j] \text{ OR *no writemask* THEN}\)
    IF (\(EVEX.b = 1\) AND (\(SRC1 \text{ *is memory*})
      \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
      THEN DEST\([i+63:i]\) \leftarrow \text{LEFT Rotate Qwords}(SRC1\([63:0]\), imm8)
      ELSE \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
      \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
      \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST\([i+63:i]\] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
        \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
        \(\Rightarrow\) \(\Rightarrow\) \(\Rightarrow\)
      FI
  FI;
ENDFOR

DEST\([\text{MAX}_V L-1:VL]\) \leftarrow 0
VPROLVQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
            ELSE DEST[i+63:i] ← LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPROLD __m512i _mm512_rol_epi32(__m512i a, int imm);
VPROLD __m512i _mm512_mask_rol_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPROLD __m512i _mm512_maskz_rol_epi32(__mmask16 k, __m512i a, int imm);
VPROLD __m256i _mm256_rol_epi32(__m256i a, int imm);
VPROLD __m256i _mm256_mask_rol_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLD __m256i _mm256_maskz_rol_epi32(__mmask8 k, __m256i a, int imm);
VPROLD __m128i _mm_rol_epi32(__m128i a, int imm);
VPROLD __m128i _mm_mask_rol_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLD __m128i _mm_maskz_rol_epi32(__mmask8 k, __m128i a, int imm);

VPROLD __m512i _mm512_rolv_epi32(__m512i a, __m512i cnt);
VPROLD __m512i _mm512_mask_rolv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPROLD __m512i _mm512_maskz_rolv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPROLD __m256i _mm256_rolv_epi32(__m256i a, __m256i cnt);
VPROLD __m256i _mm256_mask_rolv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLD __m256i _mm256_maskz_rolv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPROLD __m128i _mm_rolv_epi32(__m128i a, __m128i cnt);
VPROLD __m128i _mm_mask_rolv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLD __m128i _mm_maskz_rolv_epi32(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E4.
### Instruction Set Reference, V-Z

#### PRORD/PRORVD/PRORQ/PRORVQ—Bit Rotate Right

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 14 /r VPRORVD xmm1 [k1][z], xmm2, xmm3/m128/m32bcst</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDO.128.66.0F:W0 72 /0 ib VPRORD xmm1 [k1][z], xmm2/m128/m32bcst, imm8</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 14 /r VPRORQVQ xmm1 [k1][z], xmm2, xmm3/m128/m64bcst</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.128.66.0F:W1 72 /0 ib VPRORQ xmm1 [k1][z], xmm2/m128/m64bcst, imm8</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 14 /r VPRORVQ ymm1 [k1][z], ymm2, ymm3/m256/m32bcst</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store using result writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.256.66.0F:W0 72 /0 ib VPRORD ymm1 [k1][z], ymm2/m256/m32bcst, imm8</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 14 /r VPRORQ ymm1 [k1][z], ymm2, ymm3/m256/m64bcst</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.256.66.0F:W1 72 /0 ib VPRORQ ymm1 [k1][z], ymm2/m256/m64bcst, imm8</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 14 /r VPRORVQ zmm1 [k1][z], zmm2, zmm3/m512/m32bcst</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.512.66.0F:W0 72 /0 ib VPRORD zmm1 [k1][z], zmm2/m512/m32bcst, imm8</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 14 /r VPRORQ zmm1 [k1][z], zmm2, zmm3/m512/m64bcst</td>
<td>FV-RVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.ND.512.66.0F:W1 72 /0 ib VPRORQ zmm1 [k1][z], zmm2/m512/m64bcst, imm8</td>
<td>FV-VMI</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV-VMI</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (R)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>FV-RVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (R)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC >> COUNT) | (SRC << (32 - COUNT));

RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC >> COUNT) | (SRC << (64 - COUNT));

VPRORD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] ← RIGHT_ROTATE_DWORDS( SRC1[31:0], imm8)
    ELSE DEST[i+31:i] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VPRORVD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:j] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:j], SRC2[31:0])
            ELSE DEST[i+31:j] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:j], SRC2[i+31:j])
        FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:j] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+31:j] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0

VPRORQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC1 *is memory*)
            THEN DEST[i+63:j] ← RIGHT_ROTATE_QWORDS(SRC1[63:0], imm8)
            ELSE DEST[i+63:j] ← RIGHT_ROTATE_QWORDS(SRC1[i+63:j], imm8)
        FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:j] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+63:j] ← 0
            FI
        FI;
    ENDFOR
DEST[MAX_VL-1:VL] ← 0
VPRORVQ (EVEX encoded versions)

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

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
        ELSE DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
    FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            DEST[i+63:i] ← 0
        FI
    FI
ENDFOR

DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPRORD __m512i _mm512_ror_epi32(__m512i a, int imm);
VPRORD __m512i _mm512_mask_ror_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPRORD __m512i _mm512_maskz_ror_epi32(__mmask16 k, __m512i a, int imm);
VPRORD __m256i _mm256_ror_epi32(__m256i a, int imm);
VPRORD __m256i _mm256_mask_ror_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORD __m256i _mm256_maskz_ror_epi32(__mmask8 k, __m256i a, int imm);
VPRORD __m128i _mm_ror_epi32(__m128i a, int imm);
VPRORD __m128i _mm_mask_ror_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORD __m128i _mm_maskz_ror_epi32(__mmask8 k, __m128i a, int imm);

VPRORD __m512i _mm512_ror_epi64(__m512i a, int imm);
VPRORD __m512i _mm512_mask_ror_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPRORD __m512i _mm512_maskz_ror_epi64(__mmask8 k, __m512i a, int imm);
VPRORD __m256i _mm256_ror_epi64(__m256i a, int imm);
VPRORD __m256i _mm256_mask_ror_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORD __m256i _mm256_maskz_ror_epi64(__mmask8 k, __m256i a, int imm);
VPRORD __m128i _mm_rorv_epi64(__m128i a, __m128i cnt);
VPRORD __m128i _mm_mask_rorv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORD __m128i _mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, seeExceptions Type E4.
VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Op/En Instruction Op/En 64/32 bit Mode Support CPUID Feature Flag Description
EVEX.128.66.0F38.W0 A0 /vsib VPSCATTERDD vm32x {k1}, xmm1 T1S V/V AVX512VL AVX512F Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A0 /vsib VPSCATTERDD vm32y {k1}, ymm1 T1S V/V AVX512VL AVX512F Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A0 /vsib VPSCATTERDD vm32z {k1}, zmm1 T1S V/V AVX512F Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, xmm1 T1S V/V AVX512VL AVX512F Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, ymm1 T1S V/V AVX512VL AVX512F Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32z {k1}, zmm1 T1S V/V AVX512F Using signed dword indices, scatter qword values to memory using writemask k1.

Description
Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:
- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
• If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
• Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• Not valid with 16-bit effective addresses. Will deliver a #UD fault.
• If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.
The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.
The instruction will #UD fault if the k0 mask register is specified.
The instruction will #UD fault if EVEX.Z = 1.
**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
VPSCATTERDD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] ← SRC[i+31:i]
    k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

VPSCATTERDQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] ← SRC[i+63:i]
    k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

VPSCATTERQD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  k ← j * 64
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] ← SRC[i+31:i]
    k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

VPSCATTERQQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEX[j+63:j]) * SCALE + DISP] ← SRC[i+63:i]
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPSCATTERDD void _mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void _mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_i32scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_i64scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERQD void _mm_i64scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_mask_i64scatter_epi32(void * base, __mmask8 k, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_mask_i64scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERQD void _mm_mask_i64scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_i64scatter_epi64(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_i64scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm_i64scatter_epi64(void * base, __m128i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm512_mask_i64scatter_epi64(void * base, __mmask8 k, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_mask_i64scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm_mask_i64scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E12.
### VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 47 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.128.66.0F38.W1 47 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 47 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W1 47 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 12 /r</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 12 /r</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 12 /r</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 47 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 47 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 47 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 47 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 47 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 47 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

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

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an XMM register or a 256-bit memory location. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation
VPSLLVW (EVEX encoded version)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+15:i] ← 0
        FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
VPSLLVD (VEX.128 version)
COUNT_0 ← SRC2[31 : 0]
  (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
IF COUNT_0 < 32 THEN
  DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
  DEST[31:0] ← 0;
  (* Repeat shift operation for 2nd through 4th dwords *)
  IF COUNT_3 < 32 THEN
    DEST[127:96] ← ZeroExtend(SRC1[127:96] << COUNT_3);
  ELSE
    DEST[127:96] ← 0;
  DEST[MAX_VL-1:128] ← 0;
VPSLLVD (VEX.256 version)
COUNT_0 ← SRC2[31 : 0];
  (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 224];
IF COUNT_0 < 32 THEN
  DEST[31:0] ← ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
  DEST[31:0] ← 0;
  (* Repeat shift operation for 2nd through 7th dwords *)
  IF COUNT_7 < 32 THEN
    DEST[255:224] ← ZeroExtend(SRC1[255:224] << COUNT_7);
  ELSE
    DEST[255:224] ← 0;
  DEST[MAX_VL-1:256] ← 0;
VPSLLVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
      ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
VPSLLVQ (VEX.128 version)
COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
    DEST[63:0] ← ZeroExtend(SRC1[63:0] ◦ ◄ COUNT_0);
ELSE
    DEST[63:0] ← 0;
ENDIF;
IF COUNT_1 < 64 THEN
    DEST[127:64] ← ZeroExtend(SRC1[127:64] ◦ ◄ COUNT_1);
ELSE
    DEST[127:96] ← 0;
ENDIF;
DEST[MAX_VL-1:128] ← 0;

VPSLLVQ (VEX.256 version)
COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 *)
COUNT_3 ← SRC2[197 : 192];
IF COUNT_0 < 64 THEN
    DEST[63:0] ← ZeroExtend(SRC1[63:0] ◦ ◄ COUNT_0);
ELSE
    DEST[63:0] ← 0;
ENDIF;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
    DEST[255:192] ← ZeroExtend(SRC1[255:192] ◦ ◄ COUNT_3);
ELSE
    DEST[255:192] ← 0;
ENDIF;
DEST[MAX_VL-1:256] ← 0;

VPSLLVQ (EVEX encoded version)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] ◦ ◄ SRC2[63:0])
            ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] ◦ ◄ SRC2[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
    FI
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
Intel C/C++ Compiler Intrinsic Equivalent

VPSLLVW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i _mm256_sllv_epi16(__m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm128i_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm128i_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m256i _mm256_sllv_epi32(__m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi32(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi32(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm128i_mask_sllv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m512i _mm512_sllv_epi64(__m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_mask_sllv_epi64(__m512i s, __mmask64 k, __m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_maskz_sllv_epi64(__mmask64 k, __m512i a, __m512i cnt);
VPSLLVQ __m256i _mm256_sllv_epi64(__m256i a, __m256i cnt);
VPSLLVQ __m256i _mm256_mask_sllv_epi64(__m256i s, __mmask32 k, __m256i a, __m256i cnt);
VPSLLVQ __m256i _mm256_maskz_sllv_epi64(__mmask32 k, __m256i a, __m256i cnt);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded VPSLLVD/VPSLLVQ, see Exceptions Type E4.
EVEX-encoded VPSLLVW, see Exceptions Type E4.nb.
VPSRAVw/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 46 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.</td>
</tr>
<tr>
<td>VEX.NDS.256.66.0F38.W0 46 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 11 /r</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 11 /r</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 11 /r</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 46 /r</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 46 /r</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 46 /r</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 46 /r</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 46 /r</td>
<td>F</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 46 /r</td>
<td>F</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.</td>
</tr>
</tbody>
</table>
**Description**

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 16 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are filled with the corresponding sign bit of the source element.

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 16 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

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

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAX_VL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation**

**VPSRAVw (EVEX encoded version)**

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

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      COUNT ← SRC2[i+3:i]
      IF COUNT < 16
        THEN DEST[i+15:i] ← SignExtend(SRC1[i+15:i] >> COUNT)
        ELSE
          FOR k ← 0 TO 15
            DEST[i+k] ← SRC1[i+15]
          ENDFOR;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+15:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+15:i] ← 0
        FI
      FI
  ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
VPSRAVD (VEX.128 version)
COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAX_VL-1:128] ← 0;

VPSRAVD (VEX.256 version)
COUNT_0 ← SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 224];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAX_VL-1:256] ← 0;

VPSRAVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
i ← j * 32
IF k1[j] OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN
COUNT ← SRC2[4:0]
IF COUNT < 32
THEN
DEST[i+31:i] ← SignExtend(SRC1[i+31:i] >> COUNT)
ELSE
FOR k ← 0 TO 31
DEST[i+k] ← SRC1[i+31]
ENDFOR;
FI
ELSE
COUNT ← SRC2[i+4:i]
IF COUNT < 32
THEN
DEST[i+31:i] ← SignExtend(SRC1[i+31:i] >> COUNT)
ELSE
FOR k ← 0 TO 31
DEST[i+k] ← SRC1[i+31]
ENDFOR;
FI
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[31:0] remains unchanged* ; zeroing-masking
ELSE
DEST[31:0] ← 0
FI
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
VPSRAVQ (EVEX encoded version)

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

\[\text{FOR } j \leftarrow 0 \text{ TO } KL-1\]

\[i \leftarrow j \times 64\]

\[\text{IF } k1[j] \text{ OR *no writemask* THEN}\]

\[\text{IF (EVEX.b = 1) AND (SRC2 *is memory*)}\]

\[\text{THEN}\]

\[\text{COUNT } \leftarrow \text{SRC2}[5:0]\]

\[\text{IF COUNT } < 64\]

\[\text{THEN}\]

\[\text{DEST}[i+63:i] \leftarrow \text{SignExtend(SRC1}[i+63:i] \gg\text{ COUNT)}\]

\[\text{ELSE}\]

\[\text{FOR } k \leftarrow 0 \text{ TO } 63\]

\[\text{DEST}[i+k] \leftarrow \text{SRC1}[i+63]\]

\[\text{ENDFOR};\]

\[\text{FI}\]

\[\text{ELSE}\]

\[\text{COUNT } \leftarrow \text{SRC2}[i+5:i]\]

\[\text{IF COUNT } < 64\]

\[\text{THEN}\]

\[\text{DEST}[i+63:i] \leftarrow \text{SignExtend(SRC1}[i+63:i] \gg\text{ COUNT)}\]

\[\text{ELSE}\]

\[\text{FOR } k \leftarrow 0 \text{ TO } 63\]

\[\text{DEST}[i+k] \leftarrow \text{SRC1}[i+63]\]

\[\text{ENDFOR};\]

\[\text{FI}\]

\[\text{ELSE}\]

\[\text{IF *merging-masking* ; merging-masking}\]

\[\text{THEN *DEST}[63:0] \text{ remains unchanged*}\]

\[\text{ELSE ; zeroing-masking}\]

\[\text{DEST}[63:0] \leftarrow 0\]

\[\text{FI}\]

\[\text{ENDFOR};\]

\[\text{DEST}[\text{MAX}_\text{VL}-1:VL] \leftarrow 0;\]
Intel C/C++ Compiler Intrinsic Equivalent

VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m128i _mm128i _mm_srav_epi32(__m128i a, __m128i cnt);
VPSRAVD __m128i _mm128i _mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m128i _mm128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m512i _mm512_srav_epi64(__m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_mask_srav_epi64(__m512i s, __mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_maskz_srav_epi64(__mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m256i _mm256_srav_epi64(__m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_maskz_srav_epi64(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m128i _mm128i _mm_srav_epi64(__m128i a, __m128i cnt);
VPSRAVQ __m128i _mm128i _mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m128i _mm128i _mm_maskz_srav_epi64(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m512i _mm512_srav_epi16(__m512i a, __m512i count);
VPSRAVW __m512i _mm512_mask_srav_epi16(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_maskz_srav_epi16(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVW __m256i _mm256_srav_epi16(__m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_mask_srav_epi16(__m256i s, __mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_maskz_srav_epi16(__mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m128i _mm128i _mm_srav_epi16(__m128i a, __m128i cnt);
VPSRAVW __m128i _mm128i _mm_mask_srav_epi16(__m128i s, __mmask16 m, __m128i a, __m128i cnt);
VPSRAVW __m128i _mm128i _mm_maskz_srav_epi16(__mmask16 m, __m128i a, __m128i cnt);
VPSRAVW __m128i _mm128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.
### VPSRLVw/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.NDS.128.66.0F38.W0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 10 /r VPSRLVw xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 10 /r VPSRLVw ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 45 /r VPSRLVD zmm1, zmm2, zmm3/m512</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in 0s.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 10 /r VPSRLVw zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
**Description**
Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

**VEX.128 encoded version:** The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAX_VL-1:128) of the corresponding destination register are zeroed.

**VEX.256 encoded version:** The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAX_VL-1:256) of the corresponding ZMM register are zeroed.

**EVEX encoded VPSRLVD/Q:** The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

**EVEX encoded VPSRLVW:** The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation**

**VPSRLVW (EVEX encoded version)**

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

```plaintext
FOR j ← 0 TO KL-1
   i ← j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] ← ZeroExtend(SRC1[i+15:i] >> SRC2[i+15:i])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+15:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+15:i] ← 0
      FI
   FI
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
```

```text
VPSRLVw/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical
```
VPSRLVD (VEX.128 version)
COUNT_0 ← SRC2[31:0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127:96];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
    DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
    DEST[127:96] ← ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
    DEST[127:96] ← 0;
DEST[MAX_VL-1:128] ← 0;

VPSRLVD (VEX.256 version)
COUNT_0 ← SRC2[31:0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ← SRC2[255:224];
IF COUNT_0 < 32 THEN
    DEST[31:0] ← ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
    DEST[31:0] ← 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] ← ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
    DEST[255:224] ← 0;
DEST[MAX_VL-1:256] ← 0;

VPSRLVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j FROM 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
            ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
        FI
    FI
ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
VPSRLVQ (VEX.128 version)
COUNT_0 ← SRC2[63 : 0];
COUNT_1 ← SRC2[127 : 64];
IF COUNT_0 < 64 THEN
    DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
    DEST[63:0] ← 0;
IF COUNT_1 < 64 THEN
    DEST[127:64] ← ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
    DEST[127:64] ← 0;
DEST[MAX_VL-1:128] ← 0;

VPSRLVQ (VEX.256 version)
COUNT_0 ← SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 *)
COUNT_3 ← SRC2[255 : 192];
IF COUNT_0 < 64 THEN
    DEST[63:0] ← ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
    DEST[63:0] ← 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
    DEST[255:192] ← ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
    DEST[255:192] ← 0;
DEST[MAX_VL-1:256] ← 0;

VPSRLVQ (EVEX encoded version)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[i] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
            ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] ← 0
            FI
        FI
    ENDFOR;
DEST[MAX_VL-1:VL] ← 0;
Intel C/C++ Compiler Intrinsic Equivalent
VPSRLVW __m512i _mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_maskz_srlv_epi16( __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_srlv_epi32 (__m256i m, __m256i count)
VPSRLVW __m512i _mm512_srlv_epi32(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_maskz_srlv_epi32( __mmask16 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_mask_srlv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVW __m256i _mm256_maskz_srlv_epi32( __mmask8 k, __m256i a, __m256i cnt);
VPSRLVW __m128i _mm_mask_srlv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m128i _mm_maskz_srlv_epi32( __mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m256i _mm256_srlv_epi64 (__m256i m, __m256i count)
VPSRLVW __m128i _mm_srlv_epi32( __m128i a, __m128i cnt);
VPSRLVW __m128i _mm_srlv_epi64( __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded VPSRLVD/Q, see Exceptions Type E4.
EVEX-encoded VPSRLVw, see Exceptions Type E4.nb.
# VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

## Opcode/Description

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.DDS.128.66.0F3A.W0 25 /r ib VPTERNLOGD xmm1 [k1]{z}, xmm2, xmm3/m128/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m32bcst as source operands and writing the result to xmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F3A.W0 25 /r ib VPTERNLOGD ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m32bcst as source operands and writing the result to ymm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F3A.W0 25 /r ib VPTERNLOGD zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m32bcst as source operands and writing the result to zmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
</tr>
<tr>
<td>EVEX.DDS.128.66.0F3A.W1 25 /r ib VPTERNLOGQ xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m64bcst as source operands and writing the result to xmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
</tr>
<tr>
<td>EVEX.DDS.256.66.0F3A.W1 25 /r ib VPTERNLOGQ ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m64bcst as source operands and writing the result to ymm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
</tr>
<tr>
<td>EVEX.DDS.512.66.0F3A.W1 25 /r ib VPTERNLOGQ zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m64bcst as source operands and writing the result to zmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

## Description

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.
Table 5-11 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

**Table 5-11. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values**

<table>
<thead>
<tr>
<th>VPTERNLOGD reg1, reg2, src3, 0xE2</th>
<th>Bit Result with Imm8=0xE2</th>
<th>VPTERNLOGD reg1, reg2, src3, 0xE4</th>
<th>Bit Result with Imm8=0xE4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit(reg1)</td>
<td>Bit(reg2)</td>
<td>Bit(src3)</td>
<td>Bit(reg1)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-1 and Table 5-2 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

**Operation**

**VPTERNLOGD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
  THEN
    FOR k ← 0 TO 31
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[j][k] ← imm[(DEST[j+i] << 2) + (SRC1[i+k] << 1) + SRC2[k]]
        ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]
      Fi;
        ; table lookup of immediate bellow;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31+i] remains unchanged*;
    ELSE ; zeroing-masking
      DEST[31+i] ← 0
    Fi;
  ENDIF;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
VPTERNLOGQ (EVEX encoded versions)

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
  THEN
    FOR k ← 0 TO 63
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[k]]
        ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]
      FI;
    ENDFOR;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63+i:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[63+i:i] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

VPTERNLOGD__m512i __mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD__m512i __mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD__m512i __mm512_maskz_ternarylogic_epi32(__mmask m, __m512i a, __m512i b, int imm);
VPTERNLOGD__m256i __mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD__m256i __mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD__m256i __mm256_maskz_ternarylogic_epi32(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD__m128i __mm128_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD__m128i __mm128_mask_ternarylogic_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD__m128i __mm128_maskz_ternarylogic_epi32(__mmask8 m, __m128i a, __m128i b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.
### Opcode/ Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 26 /r VPTESTMB k2 {k1}, xmm2, xmm3/m128</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 26 /r VPTESTMB k2 {k1}, ymm2, ymm3/m256</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 26 /r VPTESTMB k2 {k1}, zmm2, zmm3/m512</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 26 /r VPTESTMW k2 {k1}, xmm2, xmm3/m128</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 26 /r VPTESTMW k2 {k1}, ymm2, ymm3/m256</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise AND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 26 /r VPTESTMW k2 {k1}, zmm2, zmm3/m512</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Bitwise AND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 27 /r VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 27 /r VPTESTMD k2 {k1}, ymm2, ymm3/m256/m32bcst</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 27 /r VPTESTMD k2 {k1}, zmm2, zmm3/m512/m32bcst</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, xmm2, xmm3/m128/m64bcst</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, ymm2, ymm3/m256/m64bcst</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, zmm2, zmm3/m512/m64bcst</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
Description
Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

VPTESTMD/VPTESTMQ: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

VPTESTMB/VPTESTMW: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

Operation

VPTESTMB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
  i ← j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;
    ELSE DEST[j] = 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPTESTw (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;
    ELSE DEST[j] = 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPTESTMD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;
        ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;
      FI;
    ELSE DEST[j] ← 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
VPTESTMQ (EVEX encoded versions)

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

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
                ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
            FI;
        ELSE DEST[j] ← 0 ; zeroing-masking only
        FI;
ENDFOR

DEST[MAX_KL-1:KL] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

VPTESTMB __mmask64 _mm512_test_epi8_mask( __m512i a, __m512i b);
VPTESTMB __mmask64 _mm512_mask_test_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_test_epi16_mask( __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_mask_test_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_test_epi32_mask( __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_mask_test_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_test_epi64_mask(__m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_mask_test_epi64_mask(__mmask8, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VPTESTMD/Q: See Exceptions Type E4.
VPTESTMB/W: See Exceptions Type E4.nb.
## VPTESTNMB/W/D/Q—Logical NAND and Set

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, xmm2, xmm3/m128</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, ymm2, ymm3/m256</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, zmm2, zmm3/m512</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512F AVX512BW</td>
<td>Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.F3.0F38.W1 26 /r VPTESTNW k2 {k1}, xmm2, xmm3/m128</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F3.0F38.W1 26 /r VPTESTNW k2 {k1}, ymm2, ymm3/m256</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F3.0F38.W1 26 /r VPTESTNW k2 {k1}, zmm2, zmm3/m512</td>
<td>FVM</td>
<td>V/V</td>
<td>AVX512F AVX512BW</td>
<td>Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, xmm2, xmm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, ymm2, ymm3/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, zmm2, zmm3/m512/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.128.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, xmm2, xmm3/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, ymm2, ymm3/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, zmm2, zmm3/m512/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F AVX512F</td>
<td>Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
</tbody>
</table>
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FVM</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

**Operation**

**VPTESTNMB**

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

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
\quad i \leftarrow j*8 \\
\quad \text{IF MaskBit}(j) \text{ OR } \text{*no writemask*} \\
\quad \quad \text{THEN} \\
\quad \quad \quad \text{DEST}[j] \leftarrow (\text{SRC1}[i+7:i] \text{ BITWISE AND SRC2}[i+7:i] == 0) ? 1 : 0 \\
\quad \quad \text{ELSE } \text{DEST}[j] \leftarrow 0; \text{zeroing masking only} \\
\quad \text{FI} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAX_KL-1:KL}] \leftarrow 0
\]

**VPTESTNMBw**

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

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
\quad i \leftarrow j*16 \\
\quad \text{IF MaskBit}(j) \text{ OR } \text{*no writemask*} \\
\quad \quad \text{THEN} \\
\quad \quad \quad \text{DEST}[j] \leftarrow (\text{SRC1}[i+15:i] \text{ BITWISE AND SRC2}[i+15:i] == 0) ? 1 : 0 \\
\quad \quad \text{ELSE } \text{DEST}[j] \leftarrow 0; \text{zeroing masking only} \\
\quad \text{FI} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAX_KL-1:KL}] \leftarrow 0
\]
VPTESTNMD
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0
                    ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0
                FI
            ELSE DEST[j] ← 0; zeroing masking only
        FI
    ENDFOR
DEST[MAX_KL-1:KL] ← 0

VPTESTNMQ
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
                    ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
                FI
            ELSE DEST[j] ← 0; zeroing masking only
        FI
    ENDFOR
DEST[MAX_KL-1:KL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VPTESTNMB __mmask64 __mm512_testn_epi8_mask(__m512i a, __m512i b);
VPTESTNMB __mmask64 __mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 __mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTESTNMB __mmask32 __mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 __mm_testn_epi8_mask(__m128i a, __m128i b);
VPTESTNMB __mmask16 __mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMB __mmask16 __mm_mask_testn_epi16_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMW __mmask32 __mm512_testn_epi16_mask(__m512i a, __m512i b);
VPTESTNMW __mmask32 __mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTNMW __mmask16 __mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTESTNMW __mmask16 __mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTESTNMW __mmask8 __mm_mask_testn_epi16_mask(__m128i a, __m128i b);
VPTESTNMW __mmask8 __mm_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMD __mmask16 __mm512_testn_epi32_mask(__m512i a, __m512i b);
VPTESTNMD __mmask16 __mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMD __mmask8 __mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTESTNMD __mmask8 __mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMD __mmask8 __mm_mask_testn_epi32_mask(__m128i a, __m128i b);
VPTESTNMD __mmask8 __mm_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMQ __mmask8 __mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTESTNMQ __mmask8 __mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMQ __mmask8 __mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTESTNMQ __mmask8 __mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMQ __mmask8 __mm_testn_epi64_mask(__m128i a, __m128i b);
VPTESTNMQ __mmask8 __mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
VPTESTNMD/VPTESTNMQ: See Exceptions Type E4.
VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb.
VRANGEPD—Range Restriction Calculation For Packed Pairs of Float64 Values

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W1 50 / r lb VRANGEPD xmm1 [k1][z], xmm2, xmm3/m128/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate two RANGE operation output value from 2 pairs of double-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 50 /r lb VRANGEPD ymm1 [k1][z], ymm2, ymm3/m256/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate four RANGE operation output value from 4 pairs of double-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 50 / r lb VRANGEPD zmm1 [k1][z], zmm2, zmm3/m512/m64bcst[sae], imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate eight RANGE operation output value from 8 pairs of double-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- **Imm8[1:0]** specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- **Imm8[3:2]** specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

**Figure 5-27.** Imm8 Controls for VRANGEPD/SD/PS/SS
When one or more of the input value is a NaN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NaN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>Result</th>
<th>IE Signaling Due to Comparison</th>
<th>Imm8[3:2] Effect to Range Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sNaN1</td>
<td>sNaN2</td>
<td>Quiet(sNaN1)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>sNaN1</td>
<td>qNaN2</td>
<td>Quiet(sNaN1)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>sNaN1</td>
<td>Norm2</td>
<td>Quiet(sNaN1)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>qNaN1</td>
<td>sNaN2</td>
<td>Quiet(sNaN2)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>qNaN1</td>
<td>qNaN2</td>
<td>qNaN1</td>
<td>No</td>
<td>Applicable</td>
</tr>
<tr>
<td>qNaN1</td>
<td>Norm2</td>
<td>Norm2</td>
<td>No</td>
<td>Applicable</td>
</tr>
<tr>
<td>Norm1</td>
<td>sNaN2</td>
<td>Quiet(sNaN2)</td>
<td>Yes</td>
<td>Ignored</td>
</tr>
<tr>
<td>Norm1</td>
<td>qNaN2</td>
<td>Norm1</td>
<td>No</td>
<td>Applicable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>Result</th>
<th>Src1</th>
<th>Src2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>-0</td>
<td>-0</td>
<td>+0</td>
<td>-0</td>
<td>+0</td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>-0</td>
<td>-0</td>
<td>+0</td>
<td>+0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>Result</th>
<th>Src1</th>
<th>Src2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

Table 5-12. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2] Table 5-13. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS and MAX, MAX_ABS Table 5-14. Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, (|a| = |b|, a>0, b<0)
**Operation**

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```c
// Check if SNAN and report IE, see also Table 5-12
IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

Src1.exp ← SRC1[62:52];
Src1.fraction ← SRC1[51:0];
IF ((Src1.exp = 0) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
  IF DAZ THEN Src1.fraction ← 0;
  ELSE IF (SRC2 <> QNAN) Set DE; FI;
FI;

Src2.exp ← SRC2[62:52];
Src2.fraction ← SRC2[51:0];
IF ((Src2.exp = 0) and (Src2.fraction != 0)) THEN// Src2 is a denormal number
  IF DAZ THEN Src2.fraction ← 0;
  ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN
  TMP[63:0] ← SRC1[63:0];
ELSE IF (SRC1 = QNAN) THEN
  TMP[63:0] ← SRC2[63:0];
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed)
  TMP[63:0] ← from Table 5-13
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01)
  TMP[63:0] ← from Table 5-14
ELSE
  Case(CmpOpCtl[1:0])
    10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
    11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
  ESAC;
FI;

Case(SignSelCtl[1:0])
  00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
  01: dest ← TMP[63:0];// Preserve sign of compare result
  10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
  11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
ESAC;
RETURN dest[63:0];
```

CmpOpCtl[1:0] = imm8[1:0];
SignSelCtl[1:0] = imm8[3:2];
VRANGEPD (EVEX encoded versions)

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

\[
\text{FOR } j \leftarrow 0 \text{ TO } KL-1 \\
i \leftarrow j \times 64
\]

\[
\text{IF k1}[j] \text{ OR *no writemask* THEN} \\
\text{IF (EVEX.b == 1) AND (SRC2 *is memory*)} \\
\text{THEN } \text{DEST}[i+63:i] \leftarrow \text{RangeDP} (\text{SRC1}[i+63:i], \text{SRC2}[63:0], \text{CmpOpCtl}[1:0], \text{SignSelCtl}[1:0]); \\
\text{ELSE } \text{DEST}[i+63:i] \leftarrow \text{RangeDP} (\text{SRC1}[i+63:i], \text{SRC2}[i+63:i], \text{DAZ}, \text{CmpOpCtl}[1:0], \text{SignSelCtl}[1:0]); \\
\text{FI;}
\]

\[
\text{ELSE} \\
\text{IF *merging-masking* ; merging-masking} \\
\text{THEN *DEST}[i+63:i] \text{ remains unchanged*} \\
\text{ELSE ; zeroing-masking} \\
\text{DEST}[i+63:i] = 0 \\
\text{FI;}
\]

\[
\text{FI;}
\]

\[
\text{ENDFOR;}
\]

\[
\text{DEST}[\text{MAX}_V L-1:V L] \leftarrow 0
\]

The following example describes a common usage of this instruction for checking that the input operand is bounded between \pm 1023.

\[
\text{VRANGEPD } zmm_\text{dst}, zmm_\text{src}, zmm_\text{1023}, 02h;
\]

Where:

\[
\text{zmm_\text{dst} is the destination operand.} \\
\text{zmm_\text{src} is the input operand to compare against } \pm 1023 \text{ (this is SRC1).} \\
\text{zmm_\text{1023} is the reference operand, contains the value of 1023 (and this is SRC2).} \\
\text{IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.}
\]

In case \(|zmm_\text{src}| < 1023 \) (i.e. SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm_\text{dst}. Otherwise, the value stored in zmm_\text{dst} will get the value of 1023 (received on zmm_\text{1023}, which is SRC2).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_\text{src}. So, even in the case of \(|zmm_\text{src}| \geq 1023, \) the selected sign of SRC1 is kept.

Thus, if zmm_\text{src} < -1023, the result of VRANGEPD will be the minimal value of -1023 while if zmm_\text{src} > +1023, the result of VRANGE will be the maximal value of +1023.
Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPD __m512d _mm512_range_pd ( __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_range_round_pd ( __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_mask_range_pd ( __m512 ds, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_mask_range_round_pd ( __m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_maskz_range_pd ( __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_maskz_range_round_pd ( __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d _mm256_range_pd ( __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_mask_range_pd ( __m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_maskz_range_pd ( __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d _mm128_range_pd ( __m128 a, __m128d b, int imm);
VRANGEPD __m128d _mm128_mask_range_pd ( __m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm128_maskz_range_pd ( __mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.
VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F3A.W0 50 /r ib</td>
<td>FV</td>
<td></td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 50 /r ib</td>
<td>FV</td>
<td></td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 50 /r ib</td>
<td>FV</td>
<td></td>
<td>AVX512DQ</td>
<td>Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst[5ae], store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.

- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.
Operation

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{

// Check if SNAN and report IE, see also Table 5-12
IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

Src1.exp ← SRC1[30:23];
Src1.fraction ← SRC1[22:0];
IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
FI;
Src2.exp ← SRC2[30:23];
Src2.fraction ← SRC2[22:0];
IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
FI;

IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-13
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-14
ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
    01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
    10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
    11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
ESAC;
FI;
Case(SignSelCtl[1:0])
00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
01: dest ← TMP[31:0];// Preserve sign of compare result
10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
ESAC;
RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
VRANGEPS

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

FOR \(j \leftarrow 0 \text{ TO KL-1}\)
    \(i \leftarrow j * 32\)
    IF \(k1[j]\) OR *no writemask* THEN
        IF (EVEX.b == 1) AND (SRC2 *is memory*)
            THEN DEST\[i+31:i\] \(\leftarrow\) RangeSP (SRC1\[i+31:i\], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
            ELSE DEST\[i+31:i\] \(\leftarrow\) RangeSP (SRC1\[i+31:i\], SRC2\[i+31:i\], DAZ, CmpOpCtl[1:0], SignSelCtl[1:0]);
        FI;
    ELSE
        IF *merging-masking* THEN *DEST\[i+31:i\] remains unchanged* ELSE ; zeroing-masking
            DEST\[i+31:i\] = 0
        FI;
    FI;
ENDFOR;

DEST[MAX_VL-1:VL] \(\leftarrow\) 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;

Where:
- zmm_dst is the destination operand.
- zmm_src is the input operand to compare against ±150.
- zmm_150 is the reference operand, contains the value of 150.
- IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |zmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 150, the selected sign of SRC1 is kept.

Thus, if zmm_src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm_src > +150, the result of VRANGE will be the maximal value of +150.
Intel C/C++ Compiler Intrinsic Equivalent
VRANGEPS _m512 _mm512_range_ps ( __m512 a, __m512 b, int imm);
VRANGEPS _m512 _mm512_range_round_ps ( __m512 a, __m512 b, int imm, int sae);
VRANGEPS _m512 _mm512_mask_range_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS _m512 _mm512_mask_range_round_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS _m512 _mm512_maskz_range_ps ( __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS _m512 _mm512_maskz_range_round_ps ( __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS _m256 _mm256_range_ps ( __m256 a, __m256 b, int imm);
VRANGEPS _m256 _mm256_mask_range_ps ( __m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS _m256 _mm256_maskz_range_ps ( __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS _m128 _mm128_range_ps ( __m128 a, __m128 b, int imm);
VRANGEPS _m128 _mm128_mask_range_ps ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS _m128 _mm128_maskz_range_ps ( __mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See Exceptions Type E2.
VRANGESD—Range Restriction Calculation From a pair of Scalar Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W1 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate a RANGE operation output value from 2 double-precision floating-point values in xmm2 and xmm3/m64, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

### Description

This instruction calculates a range operation output from two input double-precision FP values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.
**Operation**

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])

{  
  // Check if SNAN and report IE, see also Table 5-12
  IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
  IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

  Src1.exp ← SRC1[62:52];
  Src1.fraction ← SRC1[51:0];
  IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
  FI;

  Src2.exp ← SRC2[62:52];
  Src2.fraction ← SRC2[51:0];
  IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
  FI;

  IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
  ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
  ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-13
  ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-14
  ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
    01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
    10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
    11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
    ESAC;
  FI;

  Case(SignSelCtl[1:0])
  00: dest ← (SRC1[63] << 63) OR (TMP[62:0]); // Preserve Src1 sign bit
  01: dest ← TMP[63:0]; // Preserve sign of compare result
  10: dest ← (0 << 63) OR (TMP[62:0]); // Zero out sign bit
  11: dest ← (1 << 63) OR (TMP[62:0]); // Set the sign bit
  ESAC;
  RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
VRANGESD
IF k1[0] OR *no writemask*
   THEN DEST[63:0] ← RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
   IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
   ELSE ; zeroing-masking
      DEST[63:0] = 0
   FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;

Where:
xmm_dst is the destination operand.
xmm_src is the input operand to compare against ±1023.
xmm_1023 is the reference operand, contains the value of 1023.
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 1023, then its value will be written into xmm_dst. Otherwise, the value stored in xmm_dst will get the value of 1023 (received on xmm_1023).
However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 1023, the selected sign of SRC1 is kept.
Thus, if xmm_src < -1023, the result of VRANGEPOD will be the minimal value of -1023while if xmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent
VRANGESD __m128d _mm_range_sd (__m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_range_round_sd (__m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_mask_range_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_mask_range_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_maskz_range_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_maskz_range_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See Exceptions Type E3.
VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 51 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate a RANGE operation output value from 2 single-precision floating-point values in xmm2 and xmm3/m32, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction calculates a range operation output from two input single-precision FP values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

Bits 128:31 of the destination operand are copied from the respective elements of the first source operand.

When one or more of the input value is a NaN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NaN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.
VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

Operation

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-12
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[30:23];
    Src1.fraction ← SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[30:23];
    Src2.fraction ← SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-13
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-14
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
    ESAC;

    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest ← TMP[31:0];// Preserve sign of compare result
    10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0] = imm8[1:0];
SignSelCtl[1:0] = imm8[3:2];
VRANGESS

IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            DEST[31:0] = 0
    FI;
FI;

DEST[MAX_VL-1:128] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGESS zmm_dst, zmm_src, zmm_150, 02h;

Where:
xmm_dst is the destination operand.
xmm_src is the input operand to compare against ±150.
xmm_150 is the reference operand, contains the value of 150.
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in xmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 150, the selected sign of SRC1 is kept.

Thus, if xmm_src < -150, the result of VRANGESS will be the minimal value of -150 while if xmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGESS __m128 _mm_range_ss ( __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_range_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_mask_range_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_mask_range_round_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_maskz_range_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_maskz_range_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.
VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 4C /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 4C /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 4C /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocals of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand. The maximum relative error for this approximation is less than $2^{-14}$.

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

### Table 5-15. VRCP14PD/VRCP14SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \leq X \leq 2^{-1024}$</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$-2^{-1024} \leq X &lt; 0$</td>
<td>-INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$X &gt; 2^{1022}$</td>
<td>Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>$X &lt; -2^{1022}$</td>
<td>-Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>$X = 2^{-n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X = -2^{-n}$</td>
<td>-$2^n$</td>
<td></td>
</tr>
</tbody>
</table>

* in this case the mantissa is shifted right by one or two bits

Operation

**VRCP14PD** \((\text{EVEX encoded versions})\)

\((\text{KL, VL}) = (2, 128), (4, 256), (8, 512)\)

\[
\begin{align*}
\text{FOR} & \ j \leftarrow 0 \ \text{TO} \ KL-1 \\
\ & \ i \leftarrow j \times 64 \\
\ & \ \text{IF} \ k1[j] \ \text{OR} \ \text{*no writemask*} \ \text{THEN} \\
\ & \ \quad \text{IF} \ (\text{EVEX}.b = 1) \ \text{AND} \ (\text{SRC} \ \text{*is memory*}) \\
\ & \quad \quad \text{THEN} \ \text{DEST}[i+63:i] \leftarrow \text{APPROXIMATE}(1.0/\text{SRC}[63:0]); \\
\ & \quad \quad \text{ELSE} \ \text{DEST}[i+63:i] \leftarrow \text{APPROXIMATE}(1.0/\text{SRC}[i+63:i]); \\
\ & \quad \quad \text{FI;} \\
\ & \ \text{ELSE} \\
\ & \quad \text{IF} \ \text{*merging-masking*} \ \\text{; merging-masking} \\
\ & \quad \quad \text{THEN} \ \text{*DEST*[i+63:i] remains unchanged*} \\
\ & \quad \quad \text{ELSE} \ \text{; zeroing-masking} \\
\ & \quad \quad \ \text{DEST}[i+63:i] \leftarrow 0 \\
\ & \quad \text{FI;} \\
\ & \ \text{FI;} \\
\end{align*}
\]

\text{ENDFOR;} \\
\text{DEST}[\text{MAX_VL}-1:VL] \leftarrow 0

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

\[
\begin{align*}
\text{VRCP14PD } & \text{__m512d } \text{__mm512_rcp14_pd( __m512d a);} \\
\text{VRCP14PD } & \text{__m512d } \text{__mm512_mask_rcp14_pd( __m512d s, __mmask8 k, __m512d a);} \\
\text{VRCP14PD } & \text{__m512d } \text{__mm512_maskz_rcp14_pd( __mmask8 k, __m512d a);} \\
\end{align*}
\]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.
VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

### Opcode/Description

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LG.66.0F38.W1 4D / r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal of the scalar double-precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM.reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double-precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2⁻¹⁴. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-15 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


### Operation

**VRCP14SD (EVEX version)**

IF k1[0] OR *no writemask*

THEN DEST[63:0] ← APPROXIMATE(1.0/SRC2[63:0]);

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

DEST[63:0] ← 0

Fi;

DEST[127:64] ← SRC1[127:64]

DEST[MAX_VL-1:128] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**
VRCP14SD __m128d __mm_rcp14_sd(__m128d a, __m128d b);
VRCP14SD __m128d __mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d __mm_maskz_rcp14_sd(__mmask8 k, __m128d a, __m128d b);

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
See Exceptions Type E5.
VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 4C /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 4C /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 4C /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a SIMD computation of the approximate reciprocals of the packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than 2^-14.

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-16. VRCP14PS/VRCP14SS Special Cases**

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ≤ X ≤ 2^-128</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>-2^-128 ≤ X ≤ 0</td>
<td>-INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>X &gt; 2^126</td>
<td>Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>X &lt; -2^126</td>
<td>-Underflow</td>
<td>Up to 18 bits of fractions are returned*</td>
</tr>
<tr>
<td>X = 2^n</td>
<td>2^n</td>
<td></td>
</tr>
<tr>
<td>X = -2^n</td>
<td>-2^n</td>
<td></td>
</tr>
</tbody>
</table>

* in this case the mantissa is shifted right by one or two bits

Operation

VRCP14PS (EVEX encoded versions)

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

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN DEST[i+31:i] ← APPROXIMATE(1.0/SRC[31:0]);
    ELSE DEST[i+31:i] ← APPROXIMATE(1.0/SRC[i+31:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PS __m512 __mm512_rcp14_ps( __m512 a);
VRCP14PS __m512 __mm512_mask_rcp14_ps(__m512 s, __mmask16 k, __m512 a);
VRCP14PS __m512 __mm512_maskz_rcp14_ps( __mmask16 k, __m512 a);
VRCP14PS __m256 __mm256_rcp14_ps( __m256 a);
VRCP14PS __m256 __mm256_mask_rcp14_ps(__m256 s, __mmask8 k, __m256 a);
VRCP14PS __m256 __mm256_maskz_rcp14_ps( __mmask8 k, __m256 a);
VRCP14PS __m128 __mm128_rcp14_ps( __m128 a);
VRCP14PS __m128 __mm128_mask_rcp14_ps(__m128 s, __mmask8 k, __m128 a);
VRCP14PS __m128 __mm128_maskz_rcp14_ps( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.
VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| EVEX.NDS.Li.G6.66.0F38.W0 4D /r | T1S | V/V | AVX512F | Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:32]) of the first source operand (the second operand). The maximum relative error for this approximation is less than 2⁻¹⁴. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2⁻¹⁴. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-16 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


**Operation**

**VRCP14SS (EVEX version)**

IF k1[0] OR *no writemask*

THEN DEST[31:0] ← APPROXIMATE(1.0/SRC2[31:0]);

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*;

ELSE ; zeroing-masking

DEST[31:0] ← 0

FI;


DEST[MAX_VL-1:128] ← 0
Intel C/C++ Compiler Intrinsic Equivalent
VRCP14SS __m128_mm_rcp14_ss( __m128 a, __m128 b);
VRCP14SS __m128_mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRCP14SS __m128_mm_maskz_rcp14_ss( __mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E5.
VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W1 CA r VRCP28PD zmm1 [k1][z], zmm2/m512/m64bcst (sae)</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocals ($&lt;2^{-28}$ relative error) of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


### Operation

**VRCP28PD (EVEX encoded versions)**

(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN DEST[i+63:i] ← RCP_28_DP(1.0/SRC[63:0]);
            ELSE DEST[i+63:i] ← RCP_28_DP(1.0/SRC[i+63:i]);
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*;
        ELSE
            ; zeroing-masking
            DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR;
Table 5-17. VRCP28PD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2^-1022</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2^-1022 ≤ X ≤ 0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2^1022</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2^1022</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2^-n</td>
<td>2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2^-n</td>
<td>-2^n</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PD __m512d_mm512_rcp28_round_pd ( __m512d a, int sae);
VRCP28PD __m512d_mm512_mask_rcp28_round_pd(__m512d a, __mmask8 m, __m512d b, int sae);
VRCP28PD __m512d_mm512_maskz_rcp28_round_pd(__mmask8 m, __m512d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E2.
VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than \(2^{-28}\) Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W1 /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocal ((&lt; 2^{-28}) relative error) of the scalar double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than \(2^{-28}\) of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±\(\infty\), ±0.0 is returned for that element. Also, if any source element is ±0.0, ±\(\infty\) is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.


**Operation**

VRCP28SD ((EVEX encoded versions)

IF k1[0] OR *no writemask* THEN

\[ \text{DEST}[63:0] \leftarrow \text{RCP\_28\_DP}(1.0/\text{SRC2}[63:0]); \]
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

\[ \text{DEST}[63:0] \leftarrow 0 \]

Fi;

Fi;

ENDFOR;

\[ \text{DEST}[127:64] \leftarrow \text{SRC1}[127:64] \]

\[ \text{DEST}[	ext{MAX\_VL}-1:128] \leftarrow 0 \]
Table 5-18. VRCP28SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2⁻¹⁰₂²</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2⁻¹⁰₂² ≤ X ≤ 0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2¹⁰₂²</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2¹⁰₂²</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2⁻ⁿ</td>
<td>2ⁿ</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2⁻ⁿ</td>
<td>-2ⁿ</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SD __m128d _mm_rcp28_round_sd (__m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_mask_rcp28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_maskz_rcp28_round_sd(__mmask8 m, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.
**VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 CA /r VRCP28PS zmm1 k1 [x], zmm2/m512/m32bcst (sae)</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocals (&lt; 2^-28 relative error) of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results are rounded to < 2^-23 relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

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


**Operation**

**VRCP28PS (EVEX encoded versions)**

(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+31:i] ← RCP_28_SP(1.0/SRC[31:0]);
      ELSE DEST[i+31:i] ← RCP_28_SP(1.0/SRC[i+31:i]);
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*;
      ELSE ; zeroing-masking
        DEST[i+31:i] ← 0
      FI;
  FI;
ENDFOR;
Table 5-19. VRCP28PS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2(^{-126})</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2(^{-126}) ≤ X ≤ -0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2(^{126})</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2(^{126})</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2(^{-n})</td>
<td>2(^n)</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2(^{-n})</td>
<td>-2(^n)</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PS _mm512_rcp28_round_ps (__m512 a, int sae);
VRCP28PS __m512 _mm512_mask_rcp28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
VRCP28PS __m512 _mm512_maskz_rcp28_round_ps( __mmask16 m, __m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E2.
VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W0 CB /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes the approximate reciprocal ($&lt; 2^{-28}$ relative error) of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single-precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Description**

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error prior to final rounding. The final result is rounded to $< 2^{-23}$ relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is $\pm\infty$, $\pm0.0$ is returned for that element. Also, if any source element is $\pm0.0$, $\pm\infty$ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.


**Operation**

VRCP28SS ((EVEX encoded versions))

IF k1[0] OR *no writemask* THEN

DEST[31: 0] $\leftarrow$ RCP_28_SP(1.0/SRC2[31: 0]);
ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31: 0] remains unchanged*  
ELSE ; zeroing-masking

DEST[31: 0] $\leftarrow$ 0

FI;
FI;
ENDFOR;

DEST[127:32] $\leftarrow$ SRC1[127: 32]  
DEST[MAX_VL-1:128] $\leftarrow$ 0

VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error
Table 5-20. VRCP28SS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>0 ≤ X &lt; 2⁻¹²⁶</td>
<td>INF</td>
<td>Positive input denormal or zero; #Z</td>
</tr>
<tr>
<td>-2⁻¹²⁶ ≤ X ≤ 0</td>
<td>-INF</td>
<td>Negative input denormal or zero; #Z</td>
</tr>
<tr>
<td>X &gt; 2¹²⁶</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X &lt; -2¹²⁶</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = +∞</td>
<td>+0.0f</td>
<td></td>
</tr>
<tr>
<td>X = -∞</td>
<td>-0.0f</td>
<td></td>
</tr>
<tr>
<td>X = 2⁻ⁿ</td>
<td>2ⁿ</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
<tr>
<td>X = -2⁻ⁿ</td>
<td>-2ⁿ</td>
<td>Exact result (unless input/output is a denormal)</td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SS __m128 _mm_rcp28_round_ss (__m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_mask_rcp28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_maskz_rcp28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.
VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 56 / r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Perform reduction transformation on packed double-precision floating point values in xmm2/m128/m64bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 56 / r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Perform reduction transformation on packed double-precision floating point values in ymm2/m256/m64bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 56 / r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform reduction transformation on double-precision floating point values in zmm2/m512/m64bcst[sae], imm8.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM reg (w)</td>
<td>ModRM r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Perform reduction transformation of the packed binary encoded double-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is an unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[ \text{dest} = \text{src} - (\text{ROUND}(2^M \times \text{src})) \times 2^{-M}, \]

where "ROUND()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p * man2, where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0 <= |Reduced Result| <= 2^p-M-1
Then if RC != RNE: 0 <= |Reduced Result| < 2^p-M

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

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

Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS
Handling of special case of input values are listed in Table 5-21.

<table>
<thead>
<tr>
<th>[Src1] &lt; 2^{-M-1}</th>
<th>Round Mode</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RNE</td>
<td>Src1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[Src1] &lt; 2^{-M}</th>
<th>Round (Src1-2^{-M}) *</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPI, Src1 &gt; 0</td>
<td></td>
</tr>
<tr>
<td>RPI, Src1 ≤ 0</td>
<td></td>
</tr>
<tr>
<td>RNI, Src1 ≥ 0</td>
<td></td>
</tr>
<tr>
<td>RNI, Src1 &lt; 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Src1 = ±0, or Dest = ±0 (Src1!=INF)</th>
<th>NOT RNI</th>
<th>+0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1 = ±INF</td>
<td>RNI</td>
<td>-0.0</td>
</tr>
<tr>
<td>Src1 = ±NAN</td>
<td>n/a</td>
<td>QNaN(Src1)</td>
</tr>
</tbody>
</table>

* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])
{
  // Check for NaN
  IF (SRC[63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); FI;
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC source ← imm[2];
  SPE ← 0; // Suppress Precision Exception
  TMP[63:0] ← 2^{M} * ROUND(2^{M}SRC[63:0], SPE, RC_source, RC); // ROUND() treats SRC and 2^{M} as standard binary FP values
  TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RC, SPE controls
  RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}

VREDUCEPD
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC *is memory*)
      THEN DEST[i+63:j] ← ReduceArgumentDP(SRC[i+63:j], imm8[7:0]);
      ELSE DEST[i+63:j] ← ReduceArgumentDP(SRC[i+63:j], imm8[7:0]);
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:j] remains unchanged*;
      ELSE ; zeroing-masking
        DEST[i+63:j] = 0
      FI;
    FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_maskz_reduce_pd(__mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d a, int imm)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d s, __mmask8 k, __m256d a, int imm)
VREDUCEPD __m256d _mm256_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d s, __mmask8 k, __m128d a, int imm)
VREDUCEPD __m128d _mm_maskz_reduce_pd(__mmask8 k, __m128d a, int imm)

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2, additionally

#UD If EVEX.vvvv != 1111B.
VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W1 57</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512D</td>
<td>Perform a reduction transformation on a scalar double-precision floating point value in xmm3/m64 by subtracting a number of fraction bits specified by the imm8 field. Also, upper double precision floating-point value (bits[127:64]) from xmm2 are copied to xmm1[127:64]. Stores the result in xmm1 register.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Perform a reduction transformation of the binary encoded double-precision FP value in the low qword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M \times \text{src})) \times 2^{-M};$$

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,
where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent.

Then if RC = RNE: 0 <= |Reduced Result| <= 2^p-M-1
Then if RC ≠ RNE: 0 <= |Reduced Result| < 2^p-M

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 5-21.

**Operation**

ReduceArgumentDP(SRC[63:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC[63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0]; // Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0; // Suppress Precision Exception
    TMP[63:0] ← 2^M * (ROUND(2^M*SRC[63:0], SPE, RC_source, RC)); // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[63:0] ← SRC[63:0] - TMP[63:0]; // subtraction under the same RCSPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
VREDUCESD
IF k1[0] or *no writemask*
   THEN DEST[63:0] ← ReduceArgumentDP(SRC2[63:0], imm8[7:0])
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[63:0] remains unchanged* 
      ELSE ; zeroing-masking
         THEN DEST[63:0] = 0 
      FI;
   FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VREDUCESD __m128d _mm_mask_reduce_sd( __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

SIMD Floating-Point Exceptions
Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Exceptions Type E3.
**VREDECEPS—Perform Reduction Transformation on Packed Float32 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 56 /r ib VREDECEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed single-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 56 /r ib VREDECEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed single-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 56 /r ib VREDECEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sa}, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform reduction transformation on packed single-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Perform reduction transformation of the packed binary encoded single-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M \cdot \text{src})) \cdot 2^{-M},
\]

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering \( \text{src} = 2^p \cdot \text{man}2 \), where 'man2' is the normalized significand and 'p' is the unbiased exponent.

Then if RC = RNE: \( 0 \leq |\text{Reduced Result}| \leq 2^{p-M-1} \)

Then if RC ≠ RNE: \( 0 \leq |\text{Reduced Result}| < 2^{p-M} \)

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

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

Handling of special case of input values are listed in Table 5-21.
Operation

ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
  // Check for NaN
  IF (SRC[31:0] = NAN) THEN
    RETURN (Convert SRC[31:0] to QNaN); FI
  M ⇔ imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ⇔ imm8[1:0]; // Round Control for ROUND() operation
  RC source ⇔ imm[2];
  SPE ⇔ 0; // Suppress Precision Exception
  TMP[31:0] ⇔ 2^{M} * ROUND(2^{M} * SRC[31:0], SPE, RC, RC); // ROUND() treats SRC and 2^{M} as standard binary FP values
  TMP[31:0] ⇔ SRC[31:0] – TMP[31:0]; // subtraction under the same RC, SPE controls
  RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}

VREDUCEPS

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ⇔ 0 TO KL-1
  i ⇔ j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC *is memory*)
      THEN DEST[i+31:i] ⇔ ReduceArgumentSP(SRC[31:0], imm8[7:0]);
    ELSE DEST[i+31:i] ⇔ ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] = 0
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ⇔ 0

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPS __m512 _mm512_mask_reduce_ps( __m512 a, int imm, int sae)
VREDUCEPS __m512 __m512_mask_reduce_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m512 __m512_maskz_reduce_ps(__mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m256 __m256_mask_reduce_ps( __m256 a, int imm)
VREDUCEPS __m256 __m256__mask_reduce_ps(__m256 s, __mmask8 k, __m256 a, int imm)
VREDUCEPS __m256 __m256__maskz_reduce_ps(__mmask8 k, __m256 a, int imm)
VREDUCEPS __m128 __m128_mask_reduce_ps( __m128 a, int imm)
VREDUCEPS __m128 __m128_mask_reduce_ps(__m128 s, __mmask8 k, __m128 a, int imm)
VREDUCEPS __m128 __m128__maskz_reduce_ps(__mmask8 k, __m128 a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E2, additionally
#UD If EVEX.vvv != 1111B.
**VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 57 /r /lib</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform a reduction transformation on a scalar single-precision floating point value in xmm3/m32 by subtracting a number of fraction bits specified by the imm8 field. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. Stores the result in xmm1 register.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Perform a reduction transformation of the binary encoded single-precision FP value in the low dword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M*\text{src}))*2^{-M};
\]

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering \( \text{src} = 2^p \cdot \text{man2} \)

where ‘man2’ is the normalized significand and ‘p’ is the unbiased exponent.

Then if \( RC = \text{RNE} \): \( 0 \leq |\text{Reduced Result}| \leq 2^{p-M-1} \)

Then if \( RC \neq \text{RNE} \): \( 0 \leq |\text{Reduced Result}| < 2^{p-M} \)

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-21.

**Operation**

ReduceArgumentSP((SRC[31:0], imm8[7:0]))

```c
{ // Check for NaN
    IF (SRC[31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0];// Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0; // Suppress Precision Exception
    TMP[31:0] ← 2^M * ROUND(2^M*SRC[31:0], SPE, RC_source, RC); // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[31:0] ← SRC[31:0] - TMP[31:0]; // subtraction under the same RC, SPE controls
    RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```
VREDUCESS
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← ReduceArgumentSP(SRC2[31:0], imm8[7:0])
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[31:0] remains unchanged*
  ELSE ; zeroing-masking
    THEN DEST[31:0] = 0
FI;
FI;
DEST[MAX_VL-1:128] ← 0

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

VREDUCESS __m128 __mm_mask_reduce_ss( __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 __mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 __mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E3.
VRNDSCALEPD—Round Packed Float64 Values To Include A Given Number Of Fraction Bits

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 09 / r ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rounds packed double-precision floating point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 09 / r ib VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rounds packed double-precision floating point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 09 / r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rounds packed double-precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Round the double-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM/YMM/XMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPD is

\[
\text{ROUND}(x) = 2^{-M} \times \text{Round}_{\text{to}\_\text{INT}}(x \times 2^M, \text{round}\_\text{ctrl}),
\]

\[
\text{round}\_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).
VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

\[ \text{ROUND}(x) = \text{Round}_\text{to_INT}(x, \text{round}_\text{ctrl}), \]

\[ \text{round}_\text{ctrl} = \text{imm}[3:0]; \]

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

Handling of special case of input values are listed in Table 5-22.

<table>
<thead>
<tr>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Src1=±inf</strong></td>
</tr>
<tr>
<td><strong>Src1=±NAN</strong></td>
</tr>
<tr>
<td><strong>Src1=±0</strong></td>
</tr>
</tbody>
</table>

Table 5-22. VRNDSCALEPD/SD/PS/SS Special Cases

Figure 5-29. Imm8 Controls for VRNDSCALEPD/SD/PS/SS
**Operation**

RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4] ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(2^M*SRC[63:0])
    01: TMP[63:0] ← round_to_equal_or_smaller_integer(2^M*SRC[63:0])
    10: TMP[63:0] ← round_to_equal_or_larger_integer(2^M*SRC[63:0])
    11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2^M*SRC[63:0])
    ESAC

    Dest[63:0] ← 2^-M * TMP[63:0] ; scale down back to 2^-M

    if (imm8[3] = 0) Then ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
            set_precision() ; set #PE
            FI;
        FI;
    return(Dest[63:0])
}
VRNDSCALEPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF *src is a memory operand*
    THEN TMP_SRC ← BROADCAST64(SRC, VL, k1)
ELSE TMP_SRC ← SRC
FI;

FOR \( j \leftarrow 0 \) TO KL-1
    i ← j * 64
    IF k1[\( j \)] OR *no writemask*
        THEN DEST[i+63:i] ← RoundToIntegerDP((TMP_SRC[i+63:i], imm8[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN DEST[i+63:i] remains unchanged*;
        ELSE ; zeroing-masking
            DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR;

Intel C/C++ Compiler Intrinsic Equivalent
VRNDSCALEPD __m512d _mm512_roundscale_pd(__m512d a, int imm);
VRNDSCALEPD __m512d _mm512_roundscale_round_pd(__m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_mask_roundscale_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_mask_roundscale_round_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_pd(__mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_round_pd(__mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_roundscale_pd(__m256d a, int imm);
VRNDSCALEPD __m256d _mm256_mask_roundscale_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_pd(__mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m128d _mm_roundscale_pd(__m128d a, int imm);
VRNDSCALEPD __m128d _mm_mask_roundscale_pd(__mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm_maskz_roundscale_pd(__mmask8 k, __m128d a, int imm);

SIMD Floating-Point Exceptions
Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Exceptions Type E2.
VRNDSCALESD—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W1 /r ib VRNDSCALESD xmm1 {k1}[z], xmm2, xmm3/m64{sae}, imm8</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rounds scalar double-precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMrr/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

Rounds a double-precision floating-point value in the low quadword (see Figure 5-29) element the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the third operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAX_VL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESD is

\[
ROUND(x) = 2^{-M} \times \text{Round}_{\text{to} \text{INT}}(x \times 2^M, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

\[
M=\text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

\[
\text{ROUND}(x) = \text{Round}_{\text{to} \text{INT}}(x, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

**EVEX encoded version:** The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-22.
Operation

RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction  MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction  imm8[1:0] ; get round control from imm8[1:0]
    FI
    M  imm8[7:4] ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0]  round_to_nearest_even_integer(2^M*SRC[63:0])
    01: TMP[63:0]  round_to_equal_or_smaller_integer(2^M*SRC[63:0])
    10: TMP[63:0]  round_to_equal_or_larger_integer(2^M*SRC[63:0])
    11: TMP[63:0]  round_to_nearest_smallest_magnitude_integer(2^M*SRC[63:0])
    ESAC

    Dest[63:0]  2^M* TMP[63:0] ; scale down back to 2^M
    if (imm8[3] = 0) Then ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
            set precision() ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}

VRNDSCALESD (EVEX encoded version)
IF k1[0] or *no writemask*
    THEN DEST[63:0]  RoundToIntegerDP(SRC2[63:0], Zero_upper_imm[7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[63:0]  0
    FI;
    FI;
    DEST[127:64]  SRC1[127:64]
    DEST[MAX_VL-1:128]  0

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESD __m128d _mm_roundscale_sd ( __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_roundscale_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_mask_roundscale_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_mask_roundscale_round_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_maskz_roundscale_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_maskz_roundscale_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Exceptions Type E3.
VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 08 / ib VRNDSCALEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rounds packed single-precision floating point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 08 / ib VRNDSCALEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rounds packed single-precision floating point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 08 / ib VRNDSCALEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rounds packed single-precision floating-point values in zmm2/m512/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>VV</td>
</tr>
</tbody>
</table>

Description

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1' then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

\[
\text{ROUND}(x) = 2^{-M}\cdot\text{Round}_\text{to}_\text{INT}(x\cdot2^M, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x\cdot2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

\[
\text{ROUND}(x) = \text{Round}_\text{to}_\text{INT}(x, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]
Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
Handling of special case of input values are listed in Table 5-22.

### Operation

```c
RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
  if (imm8[2] = 1)
    rounding_direction ← MXCSR:RC ; get round control from MXCSR
  else
    rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
  FI
  M ← imm8[7:4] ; get the scaling factor

  case (rounding_direction)
    00: TMP[31:0] ← round_to_nearest_even_integer(2^M*SRC[31:0])
    01: TMP[31:0] ← round_to_equal_or_smaller_integer(2^M*SRC[31:0])
    10: TMP[31:0] ← round_to_equal_or_larger_integer(2^M*SRC[31:0])
    11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
  ESAC;

  Dest[31:0] ← 2^-M * TMP[31:0] ; scale down back to 2^-M
  if (imm8[3] = 0) Then ; check SPE
    if (SRC[31:0] != Dest[31:0]) Then ; check precision lost
      set_precision() ; set #PE
    FI;
  FI;
  return(Dest[31:0])
}
```

### VRNDSCALEPS (EVEX encoded versions)

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

IF *src is a memory operand*
  THEN TMP_SRC ← BROADCAST32(SRC, VL, k1)
ELSE TMP_SRC ← SRC
FI;

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← RoundToIntegerSP(TMP_SRC[i+31:i], imm8[7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VRNDSCALEPS __m512 _mm512_roundscale_ps( __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_roundscale_round_ps( __m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_ps( __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_round_ps( __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 _mm256_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m256 _mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 _mm256_maskz_roundscale_ps( __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 _mm_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m128 _mm_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 _mm_maskz_roundscale_ps( __mmask8 k, __m128 a, int imm);

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Exceptions Type E2.
VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F3A.W0 /r ib</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvv (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Rounds the single-precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The doubleword elements at bits 127:32 of the destination are copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAX_VL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to ‘1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

\[
\text{ROUND}(x) = 2^{-M}\text{Round\_to\_INT}(x*2^M, \text{round\_ctrl}),
\]

\[
\text{round\_ctrl} = \text{imm}[3:0];
\]

\[
M=\text{imm}[7:4];
\]

The operation of x*2^M is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

\[
\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),
\]

\[
\text{round\_ctrl} = \text{imm}[3:0];
\]

EVEX encoded version: The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-22.
Operation

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4] ; get the scaling factor
    case (rounding_direction)
    00: TMP[31:0] ← round_to_nearest_even_integer(2^M*SRC[31:0])
    01: TMP[31:0] ← round_to_equal_or_smaller_integer(2^M*SRC[31:0])
    10: TMP[31:0] ← round_to_equal_or_larger_integer(2^M*SRC[31:0])
    11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
    ESAC;
    Dest[31:0] ← 2^M* TMP[31:0] ; scale down back to 2^M
    if (imm8[3] = 0) Then ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then ; check precision lost
            set_precision() ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

VRNDSCALESS (EVEX encoded version)
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[31:0] remains unchanged* ; zeroing-masking
    ELSE
        THEN DEST[31:0] ← 0
    FI;
FI;
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESS __m128 _mm_roundscale_ss ( __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Exceptions Type E3.
VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 4E /r VRSQRT14PD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 4E /r VRSQRT14PD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 4E /r VRSQRT14PD zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is an XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is an XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\approx$ with the sign of the source value is returned. When the source operand is an $+\infty$ then $+\text{ZERO}$ value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

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

Operation

**VRSQRT14PD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[63:0]));
    ELSE DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+63:i]));
  FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI;
  FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

Table 5-23. VRSQRT14PD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>X = 2(^{-2^n})</td>
<td>2(^n)</td>
<td></td>
</tr>
<tr>
<td>X &lt; 0</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>X = -0</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>X = +0</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>X = +INF</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14PD __m512d __m512_rsqrt14_pd(__m512d a);
VRSQRT14PD __m512d __m512_mask_rsqrt14_pd(__m512d s, __mmask8 k, __m512d a);
VRSQRT14PD __m512d __m512_maskz_rsqrt14_pd( __mmask8 k, __m512d a);
VRSQRT14PD __m256d __mm256_rsqrt14_pd(__m256d a);
VRSQRT14PD __m256d __mm256_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d __mm256_maskz_rsqrt14_pd( __mmask8 k, __m256d a);
VRSQRT14PD __m128d __mm128_rsqrt14_pd(__m128d a);
VRSQRT14PD __m128d __mm128_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d __mm128_maskz_rsqrt14_pd( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.
VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W1 4F /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square root of the scalar double-precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Computes the approximate reciprocal of the square roots of the scalar double-precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. When the source operand is an $+\infty$ then $+\text{ZERO}$ value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


Operation

**VRSQRT14SD (EVEX version)**

IF $k_1[0]$ or *no writemask*

THEN

DEST[63:0] $\leftarrow$ APPROXIMATE$(1.0/$ SQRT(SRC2[63:0]))

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[63:0] $\leftarrow$ 0

FI;

FI;

DEST[127:64] $\leftarrow$ SRC1[127:64]

DEST[MAX_VL-1:128] $\leftarrow$ 0
Table 5-24. VRSQRT14SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>X = 2⁻ⁿ⁻１</td>
<td>2ⁿ</td>
<td></td>
</tr>
<tr>
<td>X &lt; 0</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>X = -0</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>X = +0</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>X = +INF</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

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

VRSQRT14SD __m128d _mm_rsqrt14_sd( __m128d a, __m128d b);
VRSQRT14SD __m128d _mm_mask_rsqrt14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRSQRT14SD __m128d _mm_maskz_rsqrt14_sd( __mmask8d m, __m128d a, __m128d b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E5.
VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 4E /r</td>
<td>V</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 4E /r</td>
<td>V</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 4E /r</td>
<td>V</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an +∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

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

Operation

**VRSQRT14PS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask* THEN
      IF (EVEX.b = 1) AND (SRC *is memory*)
         THEN DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[31:0]));
         ELSE DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+31:1]));
      FI;
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
         ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
         FI;
   FI;
ENDFOR;
DEST[MAX_VL-1:VL] ← 0

Table 5-25. VRSQRT14PS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>$X = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>$X = +0$</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>$X = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14PS __m512 _mm512_rsqrt14_ps( __m512 a);
VRSQRT14PS __m512 _mm512_mask_rsqrt14_ps(__m512 s, __mmask16 k, __m512 a);
VRSQRT14PS __m512 _mm512_maskz_rsqrt14_ps( __mmask16 k, __m512 a);
VRSQRT14PS __m256 _mm256_rsqrt14_ps( __m256 a);
VRSQRT14PS __m256 _mm256_mask_rsqrt14_ps(__m256 s, __mmask8 k, __m256 a);
VRSQRT14PS __m256 _mm256_maskz_rsqrt14_ps( __mmask8 k, __m256 a);
VRSQRT14PS __m128 _mm_rsqrt14_ps( __m128 a);
VRSQRT14PS __m128 _mm_mask_rsqrt14_ps(__m128 s, __mmask8 k, __m128 a);
VRSQRT14PS __m128 _mm_maskz_rsqrt14_ps( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.
VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W0 4F /r</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Computes the approximate reciprocal square root of the scalar single-precision floating-point value in xmm3/m32 and stores the result in the low doubleword element of xmm1 using writemask k1. Bits[127:32] of xmm2 is copied to xmm1[127:32].</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM[reg (w)]</td>
<td>VEX.vvvv</td>
<td>ModRM[r/m (r)]</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Computes the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than $2^{-14}$. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAX_VL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. When the source operand is an $\infty$, zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


**Operation**

VRSQRT14SS (EVEX version)

IF k1[0] or *no writemask*

THEN DEST[31:0] ← APPROXIMATE(1.0/ SQRT(SRC2[31:0]))

ELSE

IF *merging-masking*

THEN *DEST[31:0] remains unchanged* 

ELSE

THEN DEST[31:0] ← 0

FI;

FI;


DEST[MAX_VL-1:128] ← 0
Table 5-26. VRSQRT14SS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any denormal</td>
<td>Normal</td>
<td>Cannot generate overflow</td>
</tr>
<tr>
<td>$X = 2^{-2^n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>$X = +0$</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>$X = +\text{INF}$</td>
<td>$+0$</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SS __m128 _mm_rsqrt14_ss( __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_maskz_rsqrt14_ss( __mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.
VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W1 CC /r</td>
<td>FV/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the Reciprocal square root ($&lt;2^{-28}$ relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Description**
Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as $-\infty$, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return $-\infty$ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return $-\infty$ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1. EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.


**Operation**

**VRSQRT28PD (EVEX encoded versions)**

$$(KL, VL) = (8, 512)$$

FOR $j \leftarrow 0$ TO $KL-1$

\[ i \leftarrow j \times 64 \]

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

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

THEN DEST[i+63:j] $\leftarrow (1.0/ \text{SQRT}(\text{SRC}[i+63:j]));$

ELSE DEST[i+63:j] $\leftarrow (1.0/ \text{SQRT}(\text{SRC}[i+63:j]));$

FI;

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:j] $\leftarrow 0$

FI;

FI;

ENDFOR;
**Table 5-27. VRSQRT28PD Special Cases**

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNaN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>( X = 2^{-2^n} )</td>
<td>( 2^n )</td>
<td></td>
</tr>
<tr>
<td>( X &lt; 0 )</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>( X = -0 ) or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>( X = +0 ) or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>( X = +INF )</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

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

- VRSQRT28PD __m512d _mm512_rsqrt28_round_pd(__m512d a, int sae);
- VRSQRT28PD __m512d _mm512_mask_rsqrt28_round_pd(__m512d s, __mmask8 m,__m512d a, int sae);
- VRSQRT28PD __m512d _mm512_maskz_rsqrt28_round_pd(__mmask8 m,__m512d a, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E2.
VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LG.66.0F38.W1 CD /r VRSQRT28SD xmm1 [k1]{z}, xmm2, xmm3/m64 {sae}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximate reciprocal square root ($&lt;2^{-28}$ relative error) of the scalar double-precision floating-point value from xmm3/m64 and stores result in xmm1 with writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than $2^{-28}$ of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as $-\infty$, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return $-\infty$ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return $-\infty$ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.


Operation

VRSQRT28SD (EVEX encoded versions)

IF k1[0] OR *no writemask* THEN
DEST[63: 0] $\leftarrow$ (1.0/ SQRT(SRC[63: 0]));
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[63: 0] remains unchanged*
ELSE ; zeroing-masking
DEST[63: 0] $\leftarrow$ 0
FI;
FI;
ENDFOR;
DEST[127:64] $\leftarrow$ SRC[127: 64]
DEST[MAX_VL-1:128] $\leftarrow$ 0
Table 5-28. VRSQRT28SD Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>$x = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$x &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$x = -0$ or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$x = +0$ or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$x = +\infty$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

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

VRSQRT28SD __m128d __mm_rsqrt28_round_sd(__m128d a, __m128d b, int sae);
VRSQRT28SD __m128d __mm_mask_rsqrt28_round_pd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRSQRT28SD __m128d __mm_maskz_rsqrt28_round_pd(__mmask8 m, __m128d a, __m128d b, int sae);

**SIMD Floating-Point Exceptions**

Invalid (if SNaN input), Divide-by-zero

**Other Exceptions**

See Exceptions Type E3.
VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than $2^{-28}$ Relative Error

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 CC /r VRSQRT28PS zmm1 {k1}[z], zmm2/m512/m32bcst {sae}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512ER</td>
<td>Computes approximations to the Reciprocal square root ($&lt;2^{-28}$ relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores result in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Operation**

**VRSQRT28PS (EVEX encoded versions)**

\[(KL, VL) = (16, 512)\]

FOR \(j \leftarrow 0\) TO \(KL-1\)

\[i \leftarrow j \times 32\]

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

\[\text{IF (EVEX.b = 1) AND (SRC *is memory*)}
\]

THEN \(\text{DEST}[i+31:i] \leftarrow (1.0 / \text{SQRT} \text{SRC}[i+31:i])\);

ELSE \(\text{DEST}[i+31:i] \leftarrow (1.0 / \text{SQRT} \text{SRC}[i+31:i])\);

FI;

ELSE

\[\text{IF *merging-masking*}
\]

THEN \(\text{DEST}[i+31:i] \) remains unchanged*

ELSE \(\text{zeroing-masking}
\]

\[\text{DEST}[i+31:i] \leftarrow 0\]

FI;

FI;

ENDFOR;

---

**Description**

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than $2^{-28}$ of maximum relative error prior to final rounding. The final results is rounded to $< 2^{-23}$ relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as $\pm\infty$, return the canonical NaN and set the Invalid Flag (#I).

A value of $0$ must return $\pm\infty$ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return $\pm\infty$ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEK.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Table 5-29. VRSQRT28PS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>$X = 2^{-2n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$ or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +0$ or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>$X = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28PS __m512 _mm512_rsqrt28_round_ps(__m512 a, int sae);
VRSQRT28PS __m512 _mm512_mask_rsqrt28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
VRSQRT28PS __m512 _mm512_maskz_rsqrt28_round_ps(__mmask16 m, __m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E2.
VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than 2^-28 Relative Error

 Opcode/ Instruction  | Op / En | 64/32 bit Mode Support | CPUID Feature Flag | Description |
 VRSQRT28SS xmm1 [k1], xmm2, xmm3/m32 (sae) | T1S | V/V | AVX512ER | Computes approximate reciprocal square root (<2^-28 relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1 with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32]. |

Description

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.


Operation

VRSQRT28SS (EVEX encoded versions)

IF k1[0] OR *no writemask* THEN
   DEST[31: 0] ← (1.0 / SQRT(SRC[31: 0]));
ELSE
   IF *merging-masking* ; merging-masking
      THEN *DEST[31: 0] remains unchanged*
   ELSE ; zeroing-masking
      DEST[31: 0] ← 0
   FI;
FI;
ENDFOR;
DEST[MAX_VL-1:128] ← 0
Table 5-30. VRSQRT28SS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAN</td>
<td>QNAN(input)</td>
<td>If (SRC = SNaN) then #I</td>
</tr>
<tr>
<td>X = 2^-2n</td>
<td>2^n</td>
<td></td>
</tr>
<tr>
<td>X &lt; 0</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>X = -0 or negative denormal</td>
<td>-INF</td>
<td>#Z</td>
</tr>
<tr>
<td>X = +0 or positive denormal</td>
<td>+INF</td>
<td>#Z</td>
</tr>
<tr>
<td>X = +INF</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SS __m128 _mm_rsqrt28_round_ss(__m128 a, __m128 b, int sae);
VRSQRT28SS __m128 _mm512_mask_rsqrt28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRSQRT28SS __m128 _mm512_maskz_rsqrt28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.
VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a floating-point scale of the packed double-precision floating-point values in the first source operand by multiplying it by 2 power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

\[ zmm1 := zmm2 \times 2^{\text{floor}(zmm3)} \]

Floor(zmm3) means maximum integer value \( \leq zmm3 \).

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

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

Handling of special-case input values are listed in Table 5-31 and Table 5-32.

Table 5-31. VSCALEFPD/SD/PS/SS Special Cases

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>Set IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>±NaN</td>
<td>+Inf</td>
<td>QNaN(Src1)</td>
</tr>
<tr>
<td>±SNaN</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
</tr>
<tr>
<td>±Inf</td>
<td>QNaN(Src2)</td>
<td>Src1</td>
</tr>
<tr>
<td>±0</td>
<td>QNaN(Src2)</td>
<td>QNaN_Indefinite</td>
</tr>
<tr>
<td>Denorm/Norm</td>
<td>QNaN(Src2)</td>
<td>±INF (Src1 sign)</td>
</tr>
</tbody>
</table>

IF either source is SNAN
YES
IF Src2 is SNAN or -INF
IF Src2 is SNAN or +INF
Compute Result
IF Src2 is SNAN
Operation
SCALE(SRC1, SRC2)
{
TMP_SRC2 ← SRC2
TMP_SRC1 ← SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
VSCALEFPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
THEN
SET_RM(EVEX.RC);
ELSE
SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
i ← j * 64
IF k1[j] OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[63:0]);
ELSE DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[i+63:i]);
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] ← 0
FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0

Table 5-32. Additional VSCALEFPD/SD Special Cases

<table>
<thead>
<tr>
<th>Special Case</th>
<th>Returned value</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>result</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; 2(^{-1074})</td>
<td>±0 or ±Min-Denormal (Src1 sign)</td>
</tr>
<tr>
<td></td>
<td>≥ 2(^{1024})</td>
<td>±INF (Src1 sign) or ±Max-normal (Src1 sign)</td>
</tr>
</tbody>
</table>
**Intel C/C++ Compiler Intrinsic Equivalent**

VSCALEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int);
VSCALEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m256d _mm256_scalef_round_pd(__m256d a, __m256d b, int);
VSCALEFPD __m256d _mm256_mask_scalef_round_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m256d _mm256_maskz_scalef_round_pd(__mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m128d _mm_scalef_round_pd(__m128d a, __m128d b, int);
VSCALEFPD __m128d _mm_mask_scalef_round_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFPD __m128d _mm_maskz_scalef_round_pd(__mmask8 k, __m128d a, __m128d b, int);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

**Other Exceptions**

See Exceptions Type E2.
VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W1 2D /r VSCALEFSD xmm1 [k1]{z}, xmm2, xmm3/m64{er}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Scale the scalar double-precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a floating-point scale of the packed double-precision floating-point value in the first source operand by multiplying it by 2 power of the double-precision floating-point value in second source operand.

The equation of this operation is given by:

\[ xmm1 := xmm2 \times 2^{\text{floor}(xmm3)} \]

Floor(xmm3) means maximum integer value \( \leq \) xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-32.

Operation

```
SCALE(SRC1, SRC2)
[
  ; Check for denormal operands
  TMP_SRC2 ← SRC2
  TMP_SRC1 ← SRC1
  IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
  IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
  /* SRC2 is a 64 bits floating-point value */
  DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
]
```
VSCALEFSD (EVEX encoded version)
IF (EVEX.b= 1) and SRC2 *is a register*
THEN
  SET_RM(EVEX.RC);
ELSE
  SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
THEN DEST[63:0] ← SCALE(SRC1[63:0], SRC2[63:0])
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      DEST[63:0] ← 0
  FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VSCALEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions
See Exceptions Type E3.
VSCALEFPS—Scale Packed Float32 Values With Float32 Values

**Operand Encoding**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.128.66.0F38.W0 2C /r VSCALEFPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F38.W0 2C /r VSCALEFPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Scale the packed single-precision values in ymm2 using floating point values from ymm3/m256/m32bcst. Under writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F38.W0 2C /r VSCALEFPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX:vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

\[ \text{zmm1} := \text{zmm2} \times 2^{\text{floor(zmm3)}}. \]

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

**EVEX.512 encoded version:** The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

**EVEX.256 encoded version:** The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

**EVEX.128 encoded version:** The first source operand is an XMM register. The second source operand is an XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is an XMM register, conditionally updated using writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-33.

**Table 5-33. Additional VSCALEFPS/SS Special Cases**

<table>
<thead>
<tr>
<th>Special Case</th>
<th>Returned value</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(</td>
<td>\text{result}</td>
</tr>
<tr>
<td></td>
<td>(</td>
<td>\text{result}</td>
</tr>
</tbody>
</table>
Operation
SCALE(SRC1, SRC2)
{
   ; Check for denormal operands
   TMP_SRC2 ← SRC2
   TMP_SRC1 ← SRC1
   IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
   IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
   /* SRC2 is a 32 bits floating-point value */
   DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

VSCALEFPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
   THEN
      SET_RM(EVEX.RC);
   ELSE
      SET_RM(MXCSR.RM);
   FI;
FOR j ← 0 TO KL-1
   i ← j * 32
   IF k1[j] OR *no writemask* THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
         THEN DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[31:0]);
         ELSE DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[i+31:i]);
      FI;
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+31:i] ← 0
      FI
   FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0;

Intel C/C++ Compiler Intrinsic Equivalent
VSCALEFPS __m512 _mm512_scalef_round_ps(__m512 a, __m512 b, int);
VSCALEFPS __m512 _mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m512 _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m256 _mm256_scalef_round_ps(__m256 a, __m256 b, int);
VSCALEFPS __m256 _mm256_mask_scalef_round_ps(__mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m256 _mm256_maskz_scalef_round_ps(__mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m128 _mm_scalef_round_ps(__m128 a, __m128 b, int);
VSCALEFPS __m128 _mm_mask_scalef_round_ps(__mmask8 k, __m128 a, __m128 b, int);
VSCALEFPS __m128 _mm_maskz_scalef_round_ps(__mmask8 k, __m128 a, __m128 b, int);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions
See Exceptions Type E2.
### VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.LIG.66.0F38.W0 2D /r VSCALEFSS xmm1 [k1]{z}, xmm2, xmm3/m32{er}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 power of the float32 value in second source operand.

The equation of this operation is given by:

\[
\text{xmm1} := \text{xmm2} \times 2^{\text{floor(xmm3)}}.
\]

Floor(xmm3) means maximum integer value \(\leq\) xmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-33.
Operation
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

VSCALEFSS (EVEX encoded version)
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] ← 0
        FI
    FI;
DEST[MAX_VL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent
VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions
See Exceptions Type E3.
VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/E</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x {k1}, xmm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y {k1}, ymm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z {k1}, zmm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, xmm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD vm32y {k1}, ymm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD vm32z {k1}, zmm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS vm64x {k1}, xmm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y {k1}, ymm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z {k1}, zmm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD vm64x {k1}, xmm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y {k1}, ymm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z {k1}, zmm1</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
</tbody>
</table>

Description
Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
• If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
• Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
• This instruction does not perform AC checks, and so will never deliver an AC fault.
• Not valid with 16-bit effective addresses. Will deliver a #UD fault.
• If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element. The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.
**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
VSCATTERDPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP] ←
            SRC[i+31:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

VSCATTERDPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP] ←
            SRC[i+63:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

VSCATTERQPS (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] ←
            SRC[i+31:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

VSCATTERQPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP] ←
            SRC[i+63:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VSCATTERDPD void _mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERDPD void _mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void _mm512_i32scatter_ps(void * base, __m128i vdx, __m512 a, int scale);
VSCATTERDPS void _mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m128i vdx, __m512 a, int scale);
VSCATTERQPD void _mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm512_i64scatter_ps(void * base, __m512i vdx, __m256 a, int scale);
VSCATTERQPS void _mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m256 a, int scale);

**SIMD Floating-Point Exceptions**

Invalid, Overflow, Underflow, Precision, Denormal

**Other Exceptions**

See Exceptions Type E12.
VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint with Intent to Write

**Description**

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

- **T0 (temporal data)**—prefetch data into the first level cache.
- **[PS data]** For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.
- **[PD data]** For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

**Operation**

`BASE_ADDR` stands for the memory operand base address (a GPR); may not exist
`VINDEX` stands for the memory operand vector of indices (a vector register)
`SCALE` stands for the memory operand scalar (1, 2, 4 or 8)
`DISP` is the optional 1, 2 or 4 byte displacement
`PREFETCH(mem, Level, State)` Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td><code>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</code></td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 C6 /vsib VSCATTERPF0DPS vm32z {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C7 /vsib VSCATTERPF0QPS vm64z {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C6 /vsib VSCATTERPF0DPD vm32y {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C7 /vsib VSCATTERPF0QPD vm64z {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.</td>
</tr>
</tbody>
</table>
VSCATTERPFODPS (EVEX encoded version)
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR

VSCATTERPFODPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR

VSCATTERPFQOQPS (EVEX encoded version)
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR

VSCATTERPFQOPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch([BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERPF0DPS void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_prefetch_i32scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_prefetch_i64scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_mask_prefetch_i64scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_prefetch_i64scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_mask_prefetch_i64scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E12NP.
VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.512.66.0F38.W0 C6 /vsib VSCATTERPF1DPS vm32z {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C7 /vsib VSCATTERPF1QPS vm64z {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C6 /vsib VSCATTERPF1DPD vm32y {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C7 /vsib VSCATTERPF1QPD vm64z {k1}</td>
<td>T1S</td>
<td>V/V</td>
<td>AVX512PF</td>
<td>Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1S</td>
<td>BaseReg (R); VSIB:base, VectorReg(R); VSIB:index</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

- T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

1. The prefetches may happen in any order (or not at all). The instruction is a hint.
2. The mask is left unchanged.
3. Not valid with 16-bit effective addresses. Will deliver a #UD fault.
4. No FP nor memory faults may be produced by this instruction.
5. Prefetches do not handle cache line splits
6. A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.
VSCATTERPF1DPS (EVEX encoded version)
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 1)
  FI;
ENDFOR

VSCATTERPF1DPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 1)
  FI;
ENDFOR

VSCATTERPF1QPS (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 1)
  FI;
ENDFOR

VSCATTERPF1QPD (EVEX encoded version)
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 64
  IF k1[j]
    Prefetch([BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 1)
  FI;
ENDFOR

Intel C/C++ Compiler Intrinsic Equivalent
VSCATTERPF1DPD void _mm512_prefetch_i32scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_prefetch_i64scatter_pd(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i64scatter_pd(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E12NP.
VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-bit Granularity

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 23 /r ib VSHUFF32X4 ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 23 /r ib VSHUFF32X4 zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 23 /r ib VSHUFF64X2 ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 23 /r ib VSHUFF64X2 zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed double-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W0 43 /r ib VSHUFI32X4 ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W0 43 /r ib VSHUFI32X4 zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.66.0F3A.W1 43 /r ib VSHUFI64X2 ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.66.0F3A.W1 43 /r ib VSHUFI64X2 zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

256-bit Version: Moves one of the two 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

512-bit Version: Moves two of the four 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.
Operation

Select2(SRC, control) {
  CASE (control[0]) OF
    0: TMP ← SRC[127:0];
    1: TMP ← SRC[255:128];
  ESAC;
  RETURN TMP
}

Select4(SRC, control) {
  CASE (control[1:0]) OF
    0: TMP ← SRC[127:0];
    1: TMP ← SRC[255:128];
    2: TMP ← SRC[383:256];
    3: TMP ← SRC[511:384];
  ESAC;
  RETURN TMP
}

VSHUFF32x4 (EVEX versions)

(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking*
        THEN DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking*
        THEN DEST[i+31:i] ← 0
      FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0
VSHUFF64x2 (EVEX 512-bit version)

\((KL, VL) = (4, 256), (8, 512)\)

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
i \leftarrow j \times 64
\]
IF (EVEX.b = 1) AND (SRC2 *is memory*)
\[
\text{THEN } \text{TMP\_SRC2}[i+63:i] \leftarrow \text{SRC2}[63:0]
\]
ELSE \(\text{TMP\_SRC2}[i+63:i] \leftarrow \text{SRC2}[i+63:i]\)
FI;
ENDFOR;

IF VL = 256
\[
\text{TMP\_DEST}[127:0] \leftarrow \text{Select2(SRC1}[255:0], \text{imm8}[0]);
\]
\[
\text{TMP\_DEST}[255:128] \leftarrow \text{Select2(SRC2}[255:0], \text{imm8}[1]);
\]
FI;

IF VL = 512
\[
\text{TMP\_DEST}[127:0] \leftarrow \text{Select4(SRC1}[511:0], \text{imm8}[1:0]);
\]
\[
\text{TMP\_DEST}[255:128] \leftarrow \text{Select4(SRC1}[511:0], \text{imm8}[3:2]);
\]
\[
\text{TMP\_DEST}[383:256] \leftarrow \text{Select4(TMP\_SRC2}[511:0], \text{imm8}[5:4]);
\]
\[
\text{TMP\_DEST}[511:384] \leftarrow \text{Select4(TMP\_SRC2}[511:0], \text{imm8}[7:6]);
\]
FI;

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
i \leftarrow j \times 64
\]
IF k1[\(j\)] OR *no writemask*
\[
\text{THEN } \text{DEST}[i+63:i] \leftarrow \text{TMP\_DEST}[i+63:i]
\]
ELSE
\[
\text{IF } *\text{merging-masking}^* \quad ; \text{merging-masking}
\]
\[
\text{THEN } *\text{DEST}[i+63:i] \text{remains unchanged}^* \quad ; \text{merging-masking}
\]
\[
\text{ELSE } *\text{zeroing-masking}^* \quad ; \text{zeroing-masking}
\]
\[
\text{THEN } \text{DEST}[i+63:i] \leftarrow 0
\]
FI
ENDFOR

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

VSHUFI32x4 (EVEX 512-bit version)

\((KL, VL) = (8, 256), (16, 512)\)

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
i \leftarrow j \times 32
\]
IF (EVEX.b = 1) AND (SRC2 *is memory*)
\[
\text{THEN } \text{TMP\_SRC2}[i+31:i] \leftarrow \text{SRC2}[31:0]
\]
ELSE \(\text{TMP\_SRC2}[i+31:i] \leftarrow \text{SRC2}[i+31:i]\)
FI;
ENDFOR;

IF VL = 256
\[
\text{TMP\_DEST}[127:0] \leftarrow \text{Select2(SRC1}[255:0], \text{imm8}[0]);
\]
\[
\text{TMP\_DEST}[255:128] \leftarrow \text{Select2(SRC2}[255:0], \text{imm8}[1]);
\]
FI;

IF VL = 512
\[
\text{TMP\_DEST}[127:0] \leftarrow \text{Select4(SRC1}[511:0], \text{imm8}[1:0]);
\]
\[
\text{TMP\_DEST}[255:128] \leftarrow \text{Select4(SRC1}[511:0], \text{imm8}[3:2]);
\]
\[
\text{TMP\_DEST}[383:256] \leftarrow \text{Select4(TMP\_SRC2}[511:0], \text{imm8}[5:4]);
\]
\[
\text{TMP\_DEST}[511:384] \leftarrow \text{Select4(TMP\_SRC2}[511:0], \text{imm8}[7:6]);
\]
FI;

FOR \(j \leftarrow 0 \text{ TO } KL-1\)
\[
i \leftarrow j \times 32
\]
VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-bit Granularity

IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
        THEN DEST[i+31:i] ← 0
    FI
FI;
ENDFOR
DEST[MAX_VL-1:VL] ← 0

VSHUFI64x2 (EVEX 512-bit version)
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
            THEN DEST[i+63:i] ← 0
        FI
    FI
ENDFOR
DEST[MAX_VL-1:VL] ← 0
Intel C/C++ Compiler Intrinsic Equivalent

VSHUFI32x4 __m512i _mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_maskz_shuffle_i32x4( __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUFI64x2 __m512i _mm512_shuffle_i64x2(__m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_mask_shuffle_i64x2(__m512i s, __mmask8 k, __m512i a, __m512i b, int imm);
VSHUFI64x2 __m512d _mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFI64x2 __m512d _mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFI64x2 __m512d __m512d __mm512_shuffle_f64x2( __m512d s, __mmask8 k, __m512d a, __m512d b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4NF.

#UD If EVEX.L'L' = 0 for VSHUFF32x4/VSHUFF64x2.
VTTESTP/VTESTPS—Packed Bit Test

**Description**

VTTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M reg field.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

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

---

#### Opcode/ Instruction
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 0E /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.</td>
</tr>
<tr>
<td>VTESTPS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0E /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.</td>
</tr>
<tr>
<td>VTESTPS ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 0F /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.</td>
</tr>
<tr>
<td>VTESTPD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0F /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.</td>
</tr>
<tr>
<td>VTESTPD ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Operation

**VTESTPS (128-bit version)**
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;

**VTESTPS (VEX.256 encoded version)**
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
    THEN ZF ← 1;
    ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
    THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;

**VTESTPD (128-bit version)**
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
IF (TEMP[63] = TEMP[127] = 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[63] = TEMP[127] = 0)
    THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;

**VTESTPD (VEX.256 encoded version)**
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
    THEN ZF ← 1;
    ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
    THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
Intel C/C++ Compiler Intrinsic Equivalent

VTESTPS

int _mm256_testz_ps (__m256 s1, __m256 s2);
int _mm256_testc_ps (__m256 s1, __m256 s2);
int _mm256_testnzc_ps (__m256 s1, __m128 s2);
int _mm_testz_ps (__m128 s1, __m128 s2);
int _mm_testc_ps (__m128 s1, __m128 s2);
int _mm_testnzc_ps (__m128 s1, __m128 s2);

VTESTPD

int _mm256_testz_pd (__m256d s1, __m256d s2);
int _mm256_testc_pd (__m256d s1, __m256d s2);
int _mm256_testnzc_pd (__m256d s1, __m256d s2);
int _mm_testz_pd (__m128d s1, __m128d s2);
int _mm_testc_pd (__m128d s1, __m128d s2);
int _mm_testnzc_pd (__m128d s1, __m128d s2);

Flags Affected
The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 4; additionally
#UD If VEX.vvvv ≠ 1111B.
If VEX.W = 1 for VTESTPS or VTESTPD.
VZEROALL—Zero All YMM Registers

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.0F.WIG 77</td>
<td>NP</td>
<td>V/V</td>
<td>AVX</td>
<td>Zero all YMM registers.</td>
</tr>
<tr>
<td>VZEROALL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The instruction zeros contents of all XMM or YMM registers.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

### Operation

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

IF (64-bit mode)

- YMM0[VLMAX-1:0] ← 0
- YMM1[VLMAX-1:0] ← 0
- YMM2[VLMAX-1:0] ← 0
- YMM3[VLMAX-1:0] ← 0
- YMM4[VLMAX-1:0] ← 0
- YMM5[VLMAX-1:0] ← 0
- YMM6[VLMAX-1:0] ← 0
- YMM7[VLMAX-1:0] ← 0
- YMM8[VLMAX-1:0] ← 0
- YMM9[VLMAX-1:0] ← 0
- YMM10[VLMAX-1:0] ← 0
- YMM11[VLMAX-1:0] ← 0
- YMM12[VLMAX-1:0] ← 0
- YMM13[VLMAX-1:0] ← 0
- YMM14[VLMAX-1:0] ← 0
- YMM15[VLMAX-1:0] ← 0

ELSE

- YMM0[VLMAX-1:0] ← 0
- YMM1[VLMAX-1:0] ← 0
- YMM2[VLMAX-1:0] ← 0
- YMM3[VLMAX-1:0] ← 0
- YMM4[VLMAX-1:0] ← 0
- YMM5[VLMAX-1:0] ← 0
- YMM6[VLMAX-1:0] ← 0
- YMM7[VLMAX-1:0] ← 0
- YMM8-15: Unmodified

FI
Intel C/C++ Compiler Intrinsic Equivalent
VZEROALL: _mm256_zeroall()

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 8.
VZEROUPPER—Zero Upper Bits of YMM Registers

**Description**

The instruction zeros the bits in position 128 and higher of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

**Operation**

**VZEROUPPER**

If (64-bit mode)

IF (64-bit mode)

\[
\begin{align*}
\text{YMM0}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM1}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM2}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM3}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM4}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM5}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM6}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM7}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM8}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM9}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM10}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM11}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM12}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM13}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM14}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM15}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\end{align*}
\]

ELSE

\[
\begin{align*}
\text{YMM0}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM1}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM2}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM3}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM4}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM5}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM6}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM7}[\text{VLMAX}-1:128] & \leftarrow 0 \\
\text{YMM8-15: unmodified} \\
\end{align*}
\]

FI
Intel C/C++ Compiler Intrinsic Equivalent
VZEROUPPER: _mm256_zeroupper()

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type 8.
WAIT/FWAIT—Wait

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9B</td>
<td>WAIT</td>
<td>NP</td>
<td>Valid</td>
<td>Valid</td>
<td>Check pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>9B</td>
<td>FWAIT</td>
<td>NP</td>
<td>Valid</td>
<td>Valid</td>
<td>Check pending unmasked floating-point exceptions.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction’s results. See the section titled “Floating-Point Exception Synchronization” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on using the WAIT/FWAIT instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

CheckForPendingUnmaskedFloatingPointExceptions;

**FPU Flags Affected**

The C0, C1, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
WBINVD—Write Back and Invalidate Cache

Description

Writes back all modified cache lines in the processor’s internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see “Serializing Instructions” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction. This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.
#UD If the LOCK prefix is used.
Real-Address Mode Exceptions

#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0) WBINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
**WRFSBASE/WRGSBASE—Write FS/GS Segment Base**

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field. The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

### Operation

FS/GS segment base address ← SRC;

### Flags Affected

None

### C/C++ Compiler Intrinsic Equivalent

- `WRFSBASE`: `void _writefsbase_u32(unsigned int);`
- `WRFSBASE`: `_writefsbase_u64(unsigned __int64);`
- `WRGSBASE`: `void _writegsbase_u32(unsigned int);`
- `WRGSBASE`: `_writegsbase_u64(unsigned __int64);`

### Protected Mode Exceptions

- #UD The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

### Real-Address Mode Exceptions

- #UD The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

### Virtual-8086 Mode Exceptions

- #UD The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

### Compatibility Mode Exceptions

- #UD The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.

---

5-570 Vol. 2C

WRFSBASE/WRGSBASE—Write FS/GS Segment Base
64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If CR4.FSGSBASE[bit 16] = 0.
If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0

#GP(0) If the source register contains a non-canonical address.
WRMSR—Write to Model Specific Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 30</td>
<td>WRMSR</td>
<td>NP</td>
<td>Valid</td>
<td>Valid</td>
<td>Write the value in EDX:EAX to MSR specified by ECX.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 35, “Model-Specific Registers (MSRs)”, in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C*, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

### Operation

```
MSR(ECX) ← EDX:EAX;
```

### Flags Affected

None.
Protected Mode Exceptions
#GP(0) If the current privilege level is not 0.
If the value in ECX specifies a reserved or unimplemented MSR address.
If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD If the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If the value in ECX specifies a reserved or unimplemented MSR address.
If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) The WRMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
WRPKRU—Write Data to User Page Key Register

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 EF</td>
<td>WRPKRU</td>
<td>NP</td>
<td>V/V</td>
<td>OSPKE</td>
<td>Writes EAX into PKRU.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Writes the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX and RAX are ignored.

Operation

IF (ECX = 0 AND EDX = 0)
    THEN PKRU ← EAX;
ELSE #GP(0);
FI;

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRPKRU: void _wrpkru(uint32_t);

Protected Mode Exceptions

#GP(0) If ECX ≠ 0.
    If EDX ≠ 0.
#UD If the LOCK prefix is used.
    If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.
XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 XACQUIRE</td>
<td>V/V</td>
<td>HLE</td>
<td>A hint used with an “XACQUIRE-enabled” instruction to start lock elision on the instruction memory operand address.</td>
</tr>
<tr>
<td>F3 XRELEASE</td>
<td>V/V</td>
<td>HLE</td>
<td>A hint used with an “XRELEASE-enabled” instruction to end lock elision on the instruction memory operand address.</td>
</tr>
</tbody>
</table>

NOTES:
1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The “MOV mem, reg” (Opcode 88H/89H) and “MOV mem, imm” (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.
Intel 64 and IA-32 Compatibility
The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.
For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to
• REPNE/REPNZ semantics for string instructions,
• Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
• Cause #UD if prepending the VEX prefix.
• Undefined for non-string instructions or other situations.
For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to
• REP/REPE/REPZ semantics for string instructions,
• Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
• Cause #UD if prepending the VEX prefix.
• Undefined for non-string instructions or other situations.

Operation
XACQUIRE
IF XACQUIRE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN
            HLE_NEST_COUNT++
            IF (HLE_NEST_COUNT = 1) THEN
                HLE_ACTIVE ← 1
                IF 64-bit mode
                    THEN
                        restartRIP ← instruction pointer of the XACQUIRE-enabled instruction
                    ELSE
                        restartEIP ← instruction pointer of the XACQUIRE-enabled instruction
                    FI;
                Enter HLE Execution (* record register state, start tracking memory state *)
            FI; (* HLE_NEST_COUNT = 1 *)
            IF ElisionBufferAvailable
                THEN
                    Allocate elision buffer
                    Record address and data for forwarding and commit checking
                    Perform elision
                ELSE
                    Perform lock acquire operation transactionally but without elision
                FI;
            ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)
                GOTO HLE_ABORT_PROCESSING
            FI;
        ELSE
            Treat instruction as non-XACQUIRE F2H prefixed legacy instruction
        FI;
    ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)
    GOTO HLE_ABORT_PROCESSING

XRELEASE

IF XRELEASE-enabled instruction
  THEN
    IF (HLE_NEST_COUNT > 0)
      THEN
        HLE_NEST_COUNT--
        IF lock address matches in elision buffer THEN
          IF lock satisfies address and value requirements THEN
            Deallocate elision buffer
          ELSE
            GOTO HLE_ABORT_PROCESSING
          FI;
        FI;
    IF (HLE_NEST_COUNT = 0)
      THEN
        IF NoAllocatedElisionBuffer
          THEN
            Try to commit transactional execution
            IF fail to commit transactional execution
              THEN
                GOTO HLE_ABORT_PROCESSING;
            ELSE (* commit success *)
              HLE_ACTIVE ← 0
            FI;
          ELSE
            GOTO HLE_ABORT_PROCESSING
          FI;
        Fi; (* HLE_NEST_COUNT > 0 *)
      ELSE
        Treat instruction as non-XRELEASE F3H prefixed legacy instruction
      Fi;
    Fi;
  FI; (* For any HLE abort condition encountered during HLE execution *)

HLE_ABORT_PROCESSING:
  HLE_ACTIVE ← 0
  HLE_NEST_COUNT ← 0
  Restore architectural register state
  Discard memory updates performed in transaction
  Free any allocated lock elision buffers
  IF 64-bit mode
    THEN
      RIP ← restartRIP
    ELSE
      EIP ← restartEIP
  Fi;
  Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END
SIMD Floating-Point Exceptions
None

Other Exceptions
#GP(0)  If the use of prefix causes instruction length to exceed 15 bytes.
XABORT — Transactional Abort

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6 F8 ib XABORT imm8</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Causes an RTM abort if in RTM execution</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>imm8</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

Operation

XABORT

IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
Fi;

(* For any RTM abort condition encountered during RTM execution *)

RTM_ABORT_PROCESSING:
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT ← 0;
    RTM_ACTIVE ← 0;
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP;
        ELSE
            EIP ← fallbackEIP;
        Fi;
    END

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XABORT:     _xabort( unsigned int);
Other Exceptions

#UD

CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.

If LOCK prefix is used.
XADD—Exchange and Add

### Description
Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

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

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

### IA-32 Architecture Compatibility
IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

### Operation
\[
\text{TEMP} \leftarrow \text{SRC} + \text{DEST}; \\
\text{SRC} \leftarrow \text{DEST}; \\
\text{DEST} \leftarrow \text{TEMP};
\]

### Flags Affected
The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

### Protected Mode Exceptions
- **#GP(0)**: If the destination is located in a non-writable segment.
- **#SS(0)**: If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)**: If a page fault occurs.
- **#AC(0)**: If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD**: If the LOCK prefix is used but the destination is not a memory operand.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>ModRM:r/m (r, w)</td>
<td>ModRM:reg (r, w)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

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

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C7 F8 XBEGIN rel16</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.</td>
</tr>
<tr>
<td>C7 F8 XBEGIN rel32</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Offset</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort.

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

**Operation**

**XBEGIN**

IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
    THEN
        RTM_NEST_COUNT++
        IF RTM_NEST_COUNT = 1 THEN
            IF 64-bit Mode
                THEN
                    fallbackRIP ← RIP + SignExtend64(IMM)
                    (* RIP is instruction following XBEGIN instruction *)
                ELSE
                    fallbackEIP ← EIP + SignExtend32(IMM)
                    (* EIP is instruction following XBEGIN instruction *)
            FI;
            IF (64-bit mode)
                THEN IF (fallbackRIP is not canonical)
                    THEN #GP(0)
                    FI;
                ELSE IF (fallbackEIP outside code segment limit)
                    THEN #GP(0)
                    FI;
            FI;
            RTM_ACTIVE ← 1
            Enter RTM Execution (* record register state, start tracking memory state*)
        FI; (* RTM_NEST_COUNT = 1 *)
ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
   GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
   Restore architectural register state
   Discard memory updates performed in transaction
   Update EAX with status
   RTM_NEST_COUNT ← 0
   RTM_ACTIVE ← 0
   IF 64-bit mode
      THEN
         RIP ← fallbackRIP
      ELSE
         EIP ← fallbackEIP
      FI;
   END

Flags Affected
None

Intel C/C++ Compiler Intrinsic Equivalent
XBEGIN:   unsigned int _xbegin( void );

SIMD Floating-Point Exceptions
None

Protected Mode Exceptions
#UD   CPUD.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
      If LOCK prefix is used.
#GP(0) If the fallback address is outside the CS segment.

Real-Address Mode Exceptions
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.
#UD   CPUD.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
      If LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.
#UD   CPUD.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
      If LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
**64-bit Mode Exceptions**

- **#UD**
  
  CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
  
  If LOCK prefix is used.

- **#GP(0)**
  
  If the fallback address is non-canonical.
XCHG—Exchange Register/Memory with Register

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>90+rw</td>
<td>XCHG AX, r16</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r16 with AX.</td>
</tr>
<tr>
<td>90+rw</td>
<td>XCHG AX, r16</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange AX with r16.</td>
</tr>
<tr>
<td>90+rd</td>
<td>XCHG EAX, r32</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r32 with EAX.</td>
</tr>
<tr>
<td>REX.W + 90+rd</td>
<td>XCHG RAX, r64</td>
<td>O</td>
<td>N.E.</td>
<td>Exchange r64 with RAX.</td>
<td></td>
</tr>
<tr>
<td>90+rd</td>
<td>XCHG r32, EAX</td>
<td>O</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange EAX with r32.</td>
</tr>
<tr>
<td>REX.W + 90+rd</td>
<td>XCHG r64, RAX</td>
<td>O</td>
<td>N.E.</td>
<td>Exchange RAX with r64.</td>
<td></td>
</tr>
<tr>
<td>86 /r</td>
<td>XCHG r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r8 (byte register) with byte from r/m8.</td>
</tr>
<tr>
<td>REX + 86 /r</td>
<td>XCHG r/m8*, r8*</td>
<td>MR</td>
<td>N.E.</td>
<td>Exchange r8 (byte register) with byte from r/m8.</td>
<td></td>
</tr>
<tr>
<td>86 /r</td>
<td>XCHG r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange byte from r/m8 with r8 (byte register).</td>
</tr>
<tr>
<td>REX + 86 /r</td>
<td>XCHG r8*, r/m8*</td>
<td>RM</td>
<td>N.E.</td>
<td>Exchange byte from r/m8 with r8 (byte register).</td>
<td></td>
</tr>
<tr>
<td>87 /r</td>
<td>XCHG r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r16 with word from r/m16.</td>
</tr>
<tr>
<td>87 /r</td>
<td>XCHG r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange word from r/m16 with r16.</td>
</tr>
<tr>
<td>87 /r</td>
<td>XCHG r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r32 with doubleword from r/m32.</td>
</tr>
<tr>
<td>REX.W + 87 /r</td>
<td>XCHG r/m64, r64</td>
<td>MR</td>
<td>N.E.</td>
<td>Exchange r64 with quadword from r/m64.</td>
<td></td>
</tr>
<tr>
<td>87 /r</td>
<td>XCHG r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange doubleword from r/m32 with r32.</td>
</tr>
<tr>
<td>REX.W + 87 /r</td>
<td>XCHG r64, r/m64</td>
<td>RM</td>
<td>N.E.</td>
<td>Exchange quadword from r/m64 with r64.</td>
<td></td>
</tr>
</tbody>
</table>

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

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AX/EAX/RAX (r, w)</td>
<td>opcode + rd (r, w)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>0</td>
<td>opcode + rd (r, w)</td>
<td>AX/EAX/RAX (r, w)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM:r/m (r, w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor’s locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See “Bus Locking” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction’s default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.
NOTE
XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

Operation
TEMP ← DEST;
DEST ← SRC;
SRC ← TEMP;

Flags Affected
None.

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

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

Virtual-8086 Mode Exceptions
#GP(0)  If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)  If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)  If a page fault occurs.
#AC(0)  If alignment checking is enabled and an unaligned memory reference is made.
#UD  If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

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

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/ En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 D5 XEND</td>
<td>A</td>
<td>V/V</td>
<td>RTM</td>
<td>Specifies the end of an RTM code region.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

**Operation**

XEND

IF (RTM_ACTIVE = 0) THEN
SIGNAL #GP
ELSE
RTM_NEST_COUNT--
IF (RTM_NEST_COUNT = 0) THEN
Try to commit transaction
IF fail to commit transactional execution
THEN
GOTO RTM_ABORT_PROCESSING;
ELSE (* commit success *)
RTM_ACTIVE ← 0
FI;
FI;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:

Restore architectural register state
Discard memory updates performed in transaction
Update EAX with status
RTM_NEST_COUNT ← 0
RTM_ACTIVE ← 0
IF 64-bit Mode
THEN
RIP ← fallbackRIP
ELSE
EIP ← fallbackEIP
FI;
END
**Flags Affected**
None

**Intel C/C++ Compiler Intrinsic Equivalent**
XEND: void _xend( void );

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**

- **#UD**
  CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
  If LOCK or 66H or F2H or F3H prefix is used.

- **#GP(0)**
  If RTM_ACTIVE = 0.
XGETBV—Get Value of Extended Control Register

### Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, "Managing State Using the XSAVE Feature Set,” in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Use of any other value for ECX results in a general-protection (#GP) exception.

### Operation

\[
EDX:EAX \leftarrow XCR[ECX];
\]

### Flags Affected

None.

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

XGETBV: unsigned _int64 _xgetbv( unsigned int);

### Protected Mode Exceptions

- **#GP(0)** If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- **#UD** If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.
  - If 66H, F3H or F2H prefix is used.

### Real-Address Mode Exceptions

- **#GP(0)** If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- **#UD** If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.
  - If 66H, F3H or F2H prefix is used.
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
XLAT/XLATB—Table Look-up Translation

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D7</td>
<td>XLAT m8</td>
<td>NP</td>
<td>Valid</td>
<td>Valid</td>
<td>Set AL to memory byte DS:[(E)BX + unsigned AL].</td>
</tr>
<tr>
<td>D7</td>
<td>XLATB</td>
<td>NP</td>
<td>Valid</td>
<td>Valid</td>
<td>Set AL to memory byte DS:[(E)BX + unsigned AL].</td>
</tr>
<tr>
<td>REX.W + D7</td>
<td>XLATB</td>
<td>NP</td>
<td>Valid</td>
<td>N.E.</td>
<td>Set AL to memory byte [RBX + unsigned AL].</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operand” form and the “no-operand” form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a “short form” of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

Operation

IF AddressSize = 16 THEN
    AL ← (DS:BX + ZeroExtend(AL));
ELSE IF (AddressSize = 32) THEN
    AL ← (DS:EBX + ZeroExtend(AL)); Fl;
ELSE (AddressSize = 64) THEN
    AL ← (RBX + ZeroExtend(AL)); Fl;
ENDIF;

Flags Affected

None.

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If the DS, ES, FS, or GS register contains a NULL segment selector.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

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

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.
XOR—Logical Exclusive OR

### Instruction Encoding

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>34 ib</td>
<td>XOR AL, imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AL XOR imm8.</td>
</tr>
<tr>
<td>35 iw</td>
<td>XOR AX, imm16</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>AX XOR imm16.</td>
</tr>
<tr>
<td>35 id</td>
<td>XOR EAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>EAX XOR imm32.</td>
</tr>
<tr>
<td>REX.W + 35 id</td>
<td>XOR RAX, imm32</td>
<td>I</td>
<td>Valid</td>
<td>N.E.</td>
<td>RAX XOR imm32 (sign-extended).</td>
</tr>
<tr>
<td>80 /6 ib</td>
<td>XOR r/m8, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 XOR imm8.</td>
</tr>
<tr>
<td>REX + 80 /6 ib</td>
<td>XOR r/m8*, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 XOR imm8.</td>
</tr>
<tr>
<td>81 /6 iw</td>
<td>XOR r/m16, imm16</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 XOR imm16.</td>
</tr>
<tr>
<td>81 /6 id</td>
<td>XOR r/m32, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 XOR imm32.</td>
</tr>
<tr>
<td>REX.W + 81 /6 id</td>
<td>XOR r/m64, imm32</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 XOR imm32 (sign-extended).</td>
</tr>
<tr>
<td>83 /6 ib</td>
<td>XOR r/m16, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 XOR imm8 (sign-extended).</td>
</tr>
<tr>
<td>83 /6 ib</td>
<td>XOR r/m32, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 XOR imm8 (sign-extended).</td>
</tr>
<tr>
<td>REX.W + 83 /6 ib</td>
<td>XOR r/m64, imm8</td>
<td>MI</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 XOR imm8 (sign-extended).</td>
</tr>
<tr>
<td>30 /r</td>
<td>XOR r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m8 XOR r8.</td>
</tr>
<tr>
<td>REX + 30 /r</td>
<td>XOR r/m8*, r8*</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m8 XOR r8.</td>
</tr>
<tr>
<td>31 /r</td>
<td>XOR r/m16, r16</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m16 XOR r16.</td>
</tr>
<tr>
<td>31 /r</td>
<td>XOR r/m32, r32</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>r/m32 XOR r32.</td>
</tr>
<tr>
<td>REX.W + 31 /r</td>
<td>XOR r/m64, r64</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>r/m64 XOR r64.</td>
</tr>
<tr>
<td>32 /r</td>
<td>XOR r8, r/m8</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r8 XOR r/m8.</td>
</tr>
<tr>
<td>REX + 32 /r</td>
<td>XOR r8*, r/m8*</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r8 XOR r/m8.</td>
</tr>
<tr>
<td>33 /r</td>
<td>XOR r16, r/m16</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r16 XOR r/m16.</td>
</tr>
<tr>
<td>33 /r</td>
<td>XOR r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>r32 XOR r/m32.</td>
</tr>
<tr>
<td>REX.W + 33 /r</td>
<td>XOR r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>r64 XOR r/m64.</td>
</tr>
</tbody>
</table>

**NOTES:**

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

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRMr/m (r, w)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRMr/m (r, w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRMreg (r, w)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.
In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST ← DEST XOR SRC;

**Flags Affected**

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

**Protected Mode Exceptions**

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

**Real-Address Mode Exceptions**

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

**Virtual-8086 Mode Exceptions**

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

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.
XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>66 0F 57/r</td>
<td>RM</td>
<td>SSE2</td>
<td></td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128:66.0F:WIG 57 /r</td>
<td>VVM</td>
<td>AVX</td>
<td></td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256:66.0F:WIG 57 /r</td>
<td>VVM</td>
<td>AVX</td>
<td></td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128:66.0F:WT 57 /r</td>
<td>FV</td>
<td>AVX512VL, AVX512DQ</td>
<td></td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256:66.0F:WT 57 /r</td>
<td>FV</td>
<td>AVX512VL, AVX512DQ</td>
<td></td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512:66.0F:WT 57 /r</td>
<td>FV</td>
<td>AVX512DQ</td>
<td></td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

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

**EVEX.512 encoded version:** The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

**VEX.256 and EVEX.256 encoded versions:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 and EVEX.128 encoded versions:** The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

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

**Operation**

**VXORPD (EVEX encoded versions)**

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

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

VXORPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[MAX_VL-1:256] ← 0

VXORPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[MAX_VL-1:128] ← 0

XORPD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] BITWISE XOR SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE XOR SRC[127:64]
DEST[MAX_VL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent
VXORPD __m512d _mm512_xor_pd (__m512d a, __m512d b);
VXORPD __m512d _mm512_mask_xor_pd (__m512d a, __m512d m, __m512d b);
VXORPD __m512d _mm512_maskz_xor_pd (__m512d m, __m512d a);
VXORPD __m256d _mm256_xor_pd (__m256d a, __m256d b);
VXORPD __m256d _mm256_mask_xor_pd (__m256d a, __m256d m, __m256d b);
VXORPD __m256d _mm256_maskz_xor_pd (__m256d m, __m256d a);
XORPD __m128d _mm_xor_pd (__m128d a, __m128d b);
VXORPD __m128d _mm_mask_xor_pd (__m128d a, __m128d m, __m128d b);
VXORPD __m128d _mm_maskz_xor_pd (__m128d m, __m128d a);

SIMD Floating-Point Exceptions
None
Other Exceptions
Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4.
XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 57 /r</td>
<td>RM</td>
<td>V/V</td>
<td>SSE</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.NDS.128.0F.WIG 57 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.NDS.256.0F.WIG 57 /r</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.NDS.128.0F.W0 57 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.256.0F.W0 57 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.NDS.512.0F.W0 57 /r</td>
<td>FV</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RVM</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>FV</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

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

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX_VL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAX_VL-1:128) of the corresponding ZMM register destination are zeroed.

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

**VXORPS (EVEX encoded versions)**

$(K, V) = (4, 128), (8, 256), (16, 512)$

FOR $j \leftarrow 0$ TO $K - 1$

$i \leftarrow j \times 32$

IF $k_1[j]$ OR *no writemask* THEN

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

THEN $DEST[i+31:i] \leftarrow SRC1[i+31:i] \text{ BITWISE XOR } SRC2[31:0]$;

ELSE $DEST[i+31:i] \leftarrow SRC1[i+31:i] \text{ BITWISE XOR } SRC2[i+31:i]$;

FI;

ELSE

IF *merging-masking* ; merging-masking

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

ELSE

*zeroing-masking* ; zeroing-masking

$DEST[i+31:i] = 0$

FI

ENDIF

DEST[MAX_VL-1:VL] \leftarrow 0

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

DEST[31:0] \leftarrow SRC1[31:0] \text{ BITWISE XOR } SRC2[31:0]


DEST[95:64] \leftarrow SRC1[95:64] \text{ BITWISE XOR } SRC2[95:64]

DEST[127:96] \leftarrow SRC1[127:96] \text{ BITWISE XOR } SRC2[127:96]

DEST[159:128] \leftarrow SRC1[159:128] \text{ BITWISE XOR } SRC2[159:128]


DEST[255:224] \leftarrow SRC1[255:224] \text{ BITWISE XOR } SRC2[255:224]

DEST[MAX_VL-1:256] \leftarrow 0

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

DEST[31:0] \leftarrow SRC1[31:0] \text{ BITWISE XOR } SRC2[31:0]


DEST[95:64] \leftarrow SRC1[95:64] \text{ BITWISE XOR } SRC2[95:64]

DEST[127:96] \leftarrow SRC1[127:96] \text{ BITWISE XOR } SRC2[127:96]

DEST[MAX_VL-1:128] \leftarrow 0

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

DEST[31:0] \leftarrow SRC1[31:0] \text{ BITWISE XOR } SRC2[31:0]


DEST[95:64] \leftarrow SRC1[95:64] \text{ BITWISE XOR } SRC2[95:64]

DEST[127:96] \leftarrow SRC1[127:96] \text{ BITWISE XOR } SRC2[127:96]

DEST[MAX_VL-1:128] (Unmodified)

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

VXORPS __m512_mm512_xor_ps (__m512 a, __m512 b);

VXORPS __m512_mm512_mask_xor_ps (__m512 a, __mmask16 m, __m512 b);

VXORPS __m512_mm512_maskz_xor_ps (__mmask16 m, __m512 a);

VXORPS __m256_mm256_xor_ps (__m256 a, __m256 b);

VXORPS __m256_mm256_mask_xor_ps (__m256 a, __mmask8 m, __m256 b);

VXORPS __m256_mm256_maskz_xor_ps (__mmask8 m, __m256 a);

VXORPS __m128_mm_xor_ps (__m128 a, __m128 b);

VXORPS __m128_mm_mask_xor_ps (__m128 a, __mmask8 m, __m128 b);
VXORPS __m128 _mm_maskz_xor_ps (__mmask8 m, __m128 a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4.
XRSTOR—Restore Processor Extended States

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/ En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F AE /5</td>
<td>XRSTOR mem</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
<tr>
<td>REX.W+ 0F AE /5</td>
<td>XRSTOR64 mem</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.8, “Operation of XRSTOR,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If RFBM[1] = 0, XRSTOR does not update state component i.\(^1\)
- If RFBM[1] = 1 and bit \(i\) is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component \(i\).
- If RFBM[1] = 1 and XSTATE_BV[1] = 1, XRSTOR loads state component \(i\) from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVES or XSAVEOPT.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component \(i\) for which RFBM[1] = 1 and XSTATE_BV[1] = 1; it tracks as modified any state component \(i\) for which RFBM[1] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

**Operation**

\[
\begin{align*}
\text{RFBM} & \leftarrow \text{XCR0 AND EDX:EAX}; \quad /* \text{bitwise logical AND} */ \\
\text{COMPMASK} & \leftarrow \text{XCOMP_BV field from XSAVE header}; \\
\text{RSTORMASK} & \leftarrow \text{XSTATE_BV field from XSAVE header}; \\
\text{IF in VMX non-root operation} & \text{ THEN VMXNR} \leftarrow 1; \\
\end{align*}
\]

\(^1\) There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.
ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;

IF COMPMASK[63] = 0
THEN
   /* Standard form of XRSTOR */
   If RFBM[0] = 1
      THEN
         IF RSTORMASK[0] = 1
            THEN load x87 state from legacy region of XSAVE area;
            ELSE initialize x87 state;
         FI;
      FI;
   If RFBM[1] = 1
      THEN
         IF RSTORMASK[1] = 1
            THEN load XMM registers from legacy region of XSAVE area;
            ELSE set all XMM registers to 0;
         FI;
      FI;
   If RFBM[2] = 1
      THEN
         IF RSTORMASK[2] = 1
            THEN load AVX state from extended region (standard format) of XSAVE area;
            ELSE initialize AVX state;
         FI;
      FI;
   If RFBM[1] = 1 or RFBM[2] = 1
      THEN load MXCSR from legacy region of XSAVE area;
      FI;
ELSE
   /* Compacted form of XRSTOR */
   IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
      THEN /* compacted form not supported */
         #GP(0);
      FI;
   If RFBM[0] = 1
      THEN
         IF RSTORMASK[0] = 1
            THEN load x87 state from legacy region of XSAVE area;
            ELSE initialize x87 state;
         FI;
      FI;
   If RFBM[1] = 1
      THEN
         IF RSTORMASK[1] = 1
            THEN load SSE state from legacy region of XSAVE area;
            ELSE initialize SSE state;
         FI;
      FI;
   If RFBM[2] = 1
      THEN
         IF RSTORMASK[2] = 1
            THEN load AVX state from extended region (standard format) of XSAVE area;
            ELSE initialize AVX state;
         FI;
      FI;
IF RSTORMASK[2] = 1
THEN load AVX state from extended region (compacted format) of XSAVE area;
ELSE initialize AVX state;
FI;
FI;
FI;
XRSTOR_INFO ← (CPL, VMXNR, LAXA, COMP_MASK);

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XRSTOR: void _xrstor( void *, unsigned __int64);
XRSTOR: void _xrstor64( void *, unsigned __int64);

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
   If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and
   CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
   If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the
   XSTATE_BV field of the XSAVE header is 1.
   If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
   If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the
   XCOMP_BV field of the XSAVE header is 1.
   If the compacted form is executed and the bytes 63:16 of the XSAVE header are not all zero.
   If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
   operand is not aligned on a 16-byte boundary, as described above. If the alignment check
   exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
   vary with implementation, as follows. In all implementations where #AC is not signaled, a
general protection exception is signaled in its place. In addition, the width of the alignment
check may also vary with implementation. For instance, for a given implementation, an align-
ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
misalignments).

Real-Address Mode Exceptions
#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.
If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and
CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#NM
If CR0.TS[bit 3] = 1.

#UD
If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If any of the LOCK, 66H, F3H or F2H prefixes is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If a memory address is in a non-canonical form.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#NM
If CR0.TS[bit 3] = 1.

#UD
If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If any of the LOCK, 66H, F3H or F2H prefixes is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XRSTORS—Restore Processor Extended States Supervisor

**Description**

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.*

Section 13.12, "Operation of XRSTORS," of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, "Extended Region of an XSAVE Area").
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[i] = 0, XRSTORS does not update state component i.
- If RFBM[i] = 1 and bit i is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component i.
- If RFBM[i] = 1 and XSTATE_BV[i] = 1, XRSTORS loads state component i from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component i for which RFBM[i] = 1 and XSTATE_BV[i] = 1; it tracks as modified any state component i for which RFBM[i] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

**Operation**

\[
\begin{align*}
\text{RFBM} & \leftarrow (\text{XCR0 OR IA32_XSS}) \text{ AND EDX:EAX}; \\
\text{COMPMSK} & \leftarrow \text{XCOMP_BV field from XSAVE header}; \\
\text{RSTORMSK} & \leftarrow \text{XSTATE_BV field from XSAVE header}; \\
\text{IF in VMX non-root operation} & \text{ THEN VMXNR} \leftarrow 1; \\
\text{ELSE VMXNR} & \leftarrow 0; \\
\end{align*}
\]

5-606 Vol. 2C XRSTORS—Restore Processor Extended States Supervisor
LAXA ← linear address of XSAVE area;

If RFBM[0] = 1
   THEN
      IF RSTORMASK[0] = 1
         THEN load x87 state from legacy region of XSAVE area;
      ELSE initialize x87 state;
      FI;
   FI;
If RFBM[1] = 1
   THEN
      IF RSTORMASK[1] = 1
         THEN load SSE state from legacy region of XSAVE area;
      ELSE initialize SSE state;
      FI;
   FI;
If RFBM[2] = 1
   THEN
      IF RSTORMASK[2] = 1
         THEN load AVX state from extended region (compacted format) of XSAVE area;
      ELSE initialize AVX state;
      FI;
   FI;
XRSTOR_INFO ← (CPL,VMXNR,LAXA,COMPMASK);

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XRSTORS:  void _xrstors (void *, unsigned __int64);
XRSTORS64: void _xrstors64 (void *, unsigned __int64);

Protected Mode Exceptions
#GP(0)  If CPL > 0.
   If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
   If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
   If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
   If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the
   XSTATE_BV field is 1.
   If bytes 63:16 of the XSAVE header are not all zero.
   If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0)  If a memory operand effective address is outside the SS segment limit.

#PF(fault-code)  If a page fault occurs.

#NM  If CR0.TS[bit 3] = 1.

#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If any of the LOCK, 66H, F3H or F2H prefixes is used.

#AC  If this exception is disabled a general protection exception (#GP) is signaled if the memory
oprand is not aligned on a 16-byte boundary, as described above. If the alignment check
exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.
If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
If bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If any of the LOCK, 66H, F3H or F2H prefixes is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If CPL > 0.
If a memory address is in a non-canonical form.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
If bytes 63:16 of the XSAVE header are not all zero.
If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If any of the LOCK, 66H, F3H or F2H prefixes is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
**XSAVE—Save Processor Extended States**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F AE /4</td>
<td>XSAVE mem</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Save state components specified by EDX:EAX to mem.</td>
</tr>
<tr>
<td>REX.W+ 0F AE /4</td>
<td>XSAVE64 mem</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Save state components specified by EDX:EAX to mem.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.7, “Operation of XSAVE,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component \( i \) if and only if \( \text{RFBM}[i] = 1 \).
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If \( \text{RFBM}[i] = 1 \), XSAVE writes XSTATE_BV[\( i \)] with the value of XINUSE[\( i \)]. XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.” If \( \text{RFBM}[i] = 0 \), XSAVE writes XSTATE_BV[\( i \)] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

**Operation**

\[
\text{RFBM} \leftarrow \text{XCR0 AND EDX:EAX}; \quad /* \text{bitwise logical AND} */ \\
\text{OLD_BV} \leftarrow \text{XSTATE_BV field from XSAVE header};
\]

\[
\text{IF RFBM}[0] = 1 \\
\text{THEN store x87 state into legacy region of XSAVE area;}
\]

\[
\text{FI};
\]

\[
\text{IF RFBM}[1] = 1 \\
\text{THEN store XMM registers into legacy region of XSAVE area;}
\]

\[
\text{FI};
\]

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.
IF RFBM[2] = 1
    THEN store AVX state into extended region of XSAVE area;
FI;
IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

XSTATE_BV field in XSAVE header ← (OLD_BV AND ~RFBM) OR (XINUSE AND RFBM);

**Flags Affected**

None.

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

XSAVE: void _xsave( void *, unsigned __int64);
XSAVE: void _xsave64( void *, unsigned __int64);

**Protected Mode Exceptions**

<table>
<thead>
<tr>
<th>Exception Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#GP(0)</td>
<td>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</td>
</tr>
<tr>
<td>#SS(0)</td>
<td>If a memory operand effective address is outside the SS segment limit.</td>
</tr>
<tr>
<td>#PF(fault-code)</td>
<td>If a page fault occurs.</td>
</tr>
<tr>
<td>#NM</td>
<td>If CR0.TS[bit 3] = 1.</td>
</tr>
<tr>
<td>#UD</td>
<td>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</td>
</tr>
<tr>
<td></td>
<td>If CR4.OSXSAVE[bit 18] = 0.</td>
</tr>
<tr>
<td></td>
<td>If any of the LOCK, 66H, F3H or F2H prefixes is used.</td>
</tr>
</tbody>
</table>

**AC**

If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

**Real-Address Mode Exceptions**

<table>
<thead>
<tr>
<th>Exception Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#GP</td>
<td>If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.</td>
</tr>
<tr>
<td>#NM</td>
<td>If CR0.TS[bit 3] = 1.</td>
</tr>
<tr>
<td>#UD</td>
<td>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</td>
</tr>
<tr>
<td></td>
<td>If CR4.OSXSAVE[bit 18] = 0.</td>
</tr>
<tr>
<td></td>
<td>If any of the LOCK, 66H, F3H or F2H prefixes is used.</td>
</tr>
</tbody>
</table>

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.
64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form.
   If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If any of the LOCK, 66H, F3H or F2H prefixes is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
   operand is not aligned on a 16-byte boundary, as described above. If the alignment check
   exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
   vary with implementation, as follows. In all implementations where #AC is not signaled, a
general protection exception is signaled in its place. In addition, the width of the alignment
check may also vary with implementation. For instance, for a given implementation, an align-
ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
misalignments).
XSAVEC—Save Processor Extended States with Compaction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C7 /4</td>
<td>XSAVEC mem</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Save state components specified by EDX:EAX to mem with compaction.</td>
</tr>
<tr>
<td>REX.W+ 0F C7 /4</td>
<td>XSAVEC64 mem</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Save state components specified by EDX:EAX to mem with compaction.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.10, "Operation of XSAVEC," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component i if and only if RFBM[i] = 1 and XINUSE[i] = 1.¹ (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.")
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.²,³ (See Section 13.4.2, "XSAVE Header.") XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

**Operation**

\[ RFBM \leftarrow XCR0 \text{ AND EDX:EAX}; \quad /* \text{bitwise logical AND} */ \]

\[ \text{COMPMASK} \leftarrow RFBM \text{ OR } 80000000_00000000H; \]

\[
\text{IF RFBM}[0] = 1 \text{ and XINUSE}[0] = 1
\]

¹ There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.

² Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.

³ There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.
THEN store x87 state into legacy region of XSAVE area;
FI;
IF RFBM[1] = 1 and (XINUSE[1] = 1 or MXCSR ≠ 1F80H)
    THEN store SSE state into legacy region of XSAVE area;
FI;
    THEN store AVX state into extended region of XSAVE area;
FI;
XSTATE_BV field in XSAVE header ← XINUSE AND RFBM;¹
XCOMP_BV field in XSAVE header ← COMPMASK;

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEC:  void _xsavec(void *, unsigned __int64);
XSAVEC64: void _xsavec64(void *, unsigned __int64);

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
operand is not aligned on a 16-byte boundary, as described above. If the alignment check
exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
vary with implementation, as follows. In all implementations where #AC is not signaled, a
general protection exception is signaled in its place. In addition, the width of the alignment
check may also vary with implementation. For instance, for a given implementation, an align-
ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
misalignments).

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If any of the LOCK, 66H, F3H or F2H prefixes is used.

¹ If MXCSR does not have its initial value of 1F80H, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1, regardless of the value
of XINUSE[1].
Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#GP(0) If the memory address is in a non-canonical form.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If any of the LOCK, 66H, F3H or F2H prefixes is used.
#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
operand is not aligned on a 16-byte boundary, as described above. If the alignment check
exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may
vary with implementation, as follows. In all implementations where #AC is not signaled, a
general protection exception is signaled in its place. In addition, the width of the alignment
check may also vary with implementation. For instance, for a given implementation, an align-
ment check exception might be signaled for a 2-byte misalignment, whereas a general protec-
tion exception might be signaled for all other misalignments (4-, 8-, or 16-byte
misalignments).
XSAVEOPT—Save Processor Extended States Optimized

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F AE /6 XSAVEOPT</td>
<td>M</td>
<td>V/V</td>
<td>XSAVEOPT</td>
<td>Save state components specified by EDX:EAX to mem, optimizing if possible.</td>
</tr>
<tr>
<td>REX.W + 0F AE /6 XSAVEOPT64</td>
<td>M</td>
<td>V/V</td>
<td>XSAVEOPT</td>
<td>Save state components specified by EDX:EAX to mem, optimizing if possible.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.9, “Operation of XSAVEOPT,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it uses compaction and that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component \( i \) only if \( \text{RFBM}[i] = 1 \) and \( \text{XINUSE}[i] = 1 \). \(^1\) (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVEOPT may optimize and not save state component \( i \) if (1) state component \( i \) has not been modified since the last execution of XRTOR or XRSTORS; and (2) this execution of XSAVE corresponds to that last execution of XRTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If \( \text{RFBM}[i] = 1 \), XSAVEOPT writes XSTATE_BV\([i]\) with the value of XINUSE\([i]\). If \( \text{RFBM}[i] = 0 \), XSAVEOPT writes XSTATE_BV\([i]\) with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

\[
\text{RFBM} \leftarrow \text{XCR0 AND EDX:EAX}; \quad /\text{*/ bitwise logical AND */}/ \\
\text{OLD_BV} \leftarrow \text{XSTATE_BV field from XSAVE header};
\]

\(^1\) There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if \( \text{RFBM}[1] = 1 \) or \( \text{RFBM}[2] = 1 \), regardless of the value of XINUSE.
IF in VMX non-root operation
  THEN VMXNR ← 1;
  ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← 00000000_00000000H;
IF XRSTOR_INFO = ⟨CPL, VMXNR, LAXA, COMPMASK⟩
  THEN MODOPT ← 1;
  ELSE MODOPT ← 0;
FI;

IF RFBM[0] = 1 and XINUSE[0] = 1
  THEN store x87 state into legacy region of XSAVE area;
  /* might avoid saving if x87 state is not modified and MODOPT = 1 */
FI;
IF RFBM[1] = 1 and XINUSE[1]
  THEN store XMM registers into legacy region of XSAVE area;
  /* might avoid saving if XMM registers are not modified and MODOPT = 1 */
FI;
  THEN store AVX state into extended region of XSAVE area;
  /* might avoid saving if AVX state is not modified and MODOPT = 1 */
FI;
IF RFBM[1] = 1 or RFBM[2] = 1
  THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

XSTATE_BV field in XSAVE header ← (OLD_BV AND ~RFBM) OR (XINUSE AND RFBM);

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XSAVEOPT: void _xsaveopt( void *, unsigned __int64);
XSAVEOPT: void _xsaveopt64( void *, unsigned __int64);

Protected Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.
If 66H, F3H or F2H prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM  If CR0.TS[bit 3] = 1.
#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.
If 66H, F3H or F2H prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)  If the memory address is in a non-canonical form.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)  If a page fault occurs.
#NM  If CR0.TS[bit 3] = 1.
#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.
If 66H, F3H or F2H prefix is used.
XSAVES—Save Processor Extended States Supervisor

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Comp/</th>
<th>Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F C7 /5</td>
<td>XSAVES mem</td>
<td>M</td>
<td>Valid</td>
<td>Valid</td>
<td>Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.</td>
<td></td>
</tr>
<tr>
<td>REX.W+ 0F C7 /5</td>
<td>XSAVES64 mem</td>
<td>M</td>
<td>Valid</td>
<td>N.E.</td>
<td>Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>ModRM/r/m (w)</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.11, "Operation of XSAVES," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component i only if RFBM[i] = 1 and XINUSE[i] = 1.1 (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") Even if both bits are 1, XSAVES may optimize and not save state component i if (1) state component i has not been modified since the last execution of XRTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.2 (See Section 13.4.2, "XSAVE Header.") XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

---

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.

2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.
Operation

\[ \text{RFBM} \leftarrow (\text{XCR0 OR IA32_XSS}) \text{ AND } \text{EDX:EAX}; \] /* bitwise logical OR and AND */

If \text{in VMX non-root operation}

\begin{align*}
& \text{THEN } \text{VMXNR} \leftarrow 1; \\
& \text{ELSE } \text{VMXNR} \leftarrow 0;
\end{align*}

\text{FI};

\[ \text{LAXA} \leftarrow \text{linear address of XSAVE area}; \]

\[ \text{COMPMASK} \leftarrow \text{RFBM OR 80000000H}; \]

\text{IF } \text{XRSTOR_INFO} = (\text{CPL,VMXNR,LAXA,COMPMASK})

\begin{align*}
& \text{THEN } \text{MODOPT} \leftarrow 1; \\
& \text{ELSE } \text{MODOPT} \leftarrow 0;
\end{align*}

\text{FI};

\text{IF } \text{RFBM}[0] = 1 \text{ and } \text{XINUSE}[0] = 1

\begin{align*}
& \text{THEN store x87 state into legacy region of XSAVE area;} \\
& \text{/* might avoid saving if x87 state is not modified and MODOPT = 1 */}
\end{align*}

\text{FI};

\text{IF } \text{RFBM}[1] = 1 \text{ and } (\text{XINUSE}[1] = 1 \text{ or } \text{MXCSR} \neq 1F80H)

\begin{align*}
& \text{THEN store SSE state into legacy region of XSAVE area;} \\
& \text{/* might avoid saving if SSE state is not modified and MODOPT = 1 */}
\end{align*}

\text{FI};

\text{IF } \text{RFBM}[2] = 1 \text{ and } \text{XINUSE}[2] = 1

\begin{align*}
& \text{THEN store AVX state into extended region of XSAVE area;} \\
& \text{/* might avoid saving if AVX state is not modified and MODOPT = 1 */}
\end{align*}

\text{FI};

\text{XXSTATE_BV field in XSAVE header} \leftarrow \text{XINUSE AND RFBM};

\text{XXCOMP_BV field in XSAVE header} \leftarrow \text{COMPMASK};

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

\text{XSAVES: void xsaves( void *, unsigned __int64);} \hfill
\text{XSAVES64: void xsaves64( void *, unsigned __int64);} \hfill

Protected Mode Exceptions

\#GP(0) \hspace{1cm} \text{If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.}

\#SS(0) \hspace{1cm} \text{If a memory operand effective address is outside the SS segment limit.}

\#PF(fault-code) \hspace{1cm} \text{If a page fault occurs.}

\#NM \hspace{1cm} \text{If CR0.TS[bit 3] = 1.}

\#UD \hspace{1cm} \text{If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.}

\text{If CR4.OSXSAVE[bit 18] = 0.}

\text{If any of the LOCK, 66H, F3H or F2H prefixes is used.}

\#AC \hspace{1cm} \text{If this exception is disabled a general protection exception (#GP) is signaled if the memory}

\text{operand is not aligned on a 16-byte boundary, as described above. If the alignment check}

\begin{itemize}
\item[1.] \text{If MXCSR does not have its initial value of 1F80H, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1, regardless of the value of XINUSE[1].}
\end{itemize}
exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP  If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
 If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM  If CR0.TS[bit 3] = 1.

#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
 If CR4.OSXSAVE[bit 18] = 0.
 If any of the LOCK, 66H, F3H or F2H prefixes is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)  If the memory address is in a non-canonical form.
 If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#SS(0)  If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code)  If a page fault occurs.

#NM  If CR0.TS[bit 3] = 1.

#UD  If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
 If CR4.OSXSAVE[bit 18] = 0.
 If any of the LOCK, 66H, F3H or F2H prefixes is used.

#AC  If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XSETBV—Set Extended Control Register

**Description**

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, “Enabling the XSAVE Feature Set and XSAVE-Enabled Features,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

**Operation**

\[
\text{XCR[ECX]} \leftarrow \text{EDX:EAX};
\]

**Flags Affected**

None.

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

XSETBV: void _xsetbv( unsigned int, unsigned __int64);

**Protected Mode Exceptions**

- #GP(0) If the current privilege level is not 0.
  - If an invalid XCR is specified in ECX.
  - If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
  - If an attempt is made to clear bit 0 of XCR0.
  - If an attempt is made to set XCR0[2:1] to 10b.
- #UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.
  - If 66H, F3H or F2H prefix is used.
Real-Address Mode Exceptions
#GP If an invalid XCR is specified in ECX.
If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
If an attempt is made to clear bit 0 of XCR0.
If an attempt is made to set XCR0[2:1] to 10b.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.
If 66H, F3H or F2H prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) The XSETBV instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
XTEST — Test If In Transactional Execution

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 01 D6 XTEST</td>
<td>A</td>
<td>V/V</td>
<td>HLE or RTM</td>
<td>Test if executing in a transactional region</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

**Operation**

\[
\text{XTEST} \\
\text{IF (RTM\_ACTIVE = 1 OR HLE\_ACTIVE = 1)} \\
\quad \text{THEN} \\
\quad \quad \text{ZF} \leftarrow 0 \\
\quad \text{ELSE} \\
\quad \quad \text{ZF} \leftarrow 1 \\
\text{FI;}
\]

**Flags Affected**

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

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

\[
\text{XTEST: } \text{int}_{-} \text{xtest}(\text{void});
\]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

#UD CPUID.(EAX=7, ECX=0):HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0. If LOCK or 66H or F2H or F3H prefix is used.