Notices & Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

All product plans and roadmaps are subject to change without notice.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not “commercial” names and not intended to function as trademarks.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.
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 shorthand 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 of 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. Lower 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

<table>
<thead>
<tr>
<th>Imm</th>
<th>[7:4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3:0]</td>
<td>0H</td>
</tr>
<tr>
<td>00H</td>
<td>FALSE</td>
</tr>
<tr>
<td>02H</td>
<td>andCnorBA</td>
</tr>
<tr>
<td>04H</td>
<td>andBnorAC</td>
</tr>
<tr>
<td>06H</td>
<td>norAxnorBC</td>
</tr>
<tr>
<td>07H</td>
<td>norAandBC</td>
</tr>
<tr>
<td>08H</td>
<td>norAnandBC</td>
</tr>
<tr>
<td>0AH</td>
<td>andCIA</td>
</tr>
<tr>
<td>0EH</td>
<td>norAnorBC</td>
</tr>
<tr>
<td>0FH</td>
<td>IA</td>
</tr>
</tbody>
</table>
Table 5-2. Upper 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

<table>
<thead>
<tr>
<th>Imm</th>
<th>Table 5-1</th>
<th>Table 5-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3:0]</td>
<td>08H</td>
<td>09H</td>
</tr>
<tr>
<td>00H</td>
<td>andABC</td>
<td>andA$xor$BC</td>
</tr>
<tr>
<td>07H</td>
<td>xnorA$and$BC</td>
<td>A?xnorBC$and$BC</td>
</tr>
<tr>
<td>08H</td>
<td>andCB</td>
<td>A?xnorBC$and$BC</td>
</tr>
<tr>
<td>09H</td>
<td>B?C$nor$AC</td>
<td>xnorCB</td>
</tr>
<tr>
<td>0AH</td>
<td>A?andBC:CC</td>
<td>A?xnorBCCC</td>
</tr>
<tr>
<td>0BH</td>
<td>B?C:IA</td>
<td>B?C$and$AC</td>
</tr>
<tr>
<td>0CH</td>
<td>A?andBCB</td>
<td>A?xnorBCB</td>
</tr>
<tr>
<td>0DH</td>
<td>C?BIA</td>
<td>C?B$and$AC</td>
</tr>
<tr>
<td>0EH</td>
<td>A?andBCorB</td>
<td>A?xnorBCorB</td>
</tr>
<tr>
<td>0FH</td>
<td>nandA$and$BC</td>
<td>nandA$xor$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:

Result_imm8 = Table_Lookup_Entry( 0F0H, 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

<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 03 / r ib VALIGND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 03 / r ib VALIGNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 03 / r ib VALIGND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 03 / r ib VALIGNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 03 / r ib VALIGND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 03 / r ib VALIGNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg (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 (SRC2 *is memory*) (AND EVEX.b = 1)
  THEN
    FOR j := 0 TO KL-1
      i := j * 32
      src[i+31:i] := SRC2[31:0]
    ENDFOR;
    ELSE src := SRC2
  FI
  ; Concatenate sources
  tmp[VL-1:0] := src[VL-1:0]
  tmp[2VL-1:VL] := SRC1[VL-1:0]
  ; Shift right doubleword elements
  IF VL = 128
    THEN SHIFT = imm8[1:0]
  ELSE
    IF VL = 256
      THEN SHIFT = imm8[2:0]
    ELSE SHIFT = imm8[3:0]
    FI
  FI
  ; Apply writemask
  FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := tmp[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI
  ENDFOR;
  DEST[MAXVL-1:VL] := 0

DEST[MAXVL-1:VL] := 0
VALIGNQ (EVEX encoded versions)

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

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

THEN

FOR j := 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* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI
ENDFOR;
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

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_alignr_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i _mm256_mask_alignr_epi32(__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);
VALIGNQ __m512i _mm512_alignr_epi64(__m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_mask_alignr_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_maskz_alignr_epi64(__mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i _mm256_mask_alignr_epi64(__mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i _mm_mask_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i _mm_maskz_alignr_epi64(__mmask8 k, __m128i a, __m128i b, int cnt);

Exceptions

See Table 2-50, "Type E4NF Class Exception Conditions".
**VBLENDMPD/VBLENDMPS—Blend Float64/Float32 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.128.66.0F38.W1 65 /r VBLENDMPD xmm1 (k1[z]), xmm2, xmm3/m128/m64bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 65 /r VBLENDMPD ymm1 (k1[z]), ymm2, ymm3/m256/m64bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 65 /r VBLENDMPD zmm1 (k1[z]), zmm2, zmm3/m512/m64bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 65 /r VBLENDMPS xmm1 (k1[z]), xmm2, xmm3/m128/m32bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 65 /r VBLENDMPS ymm1 (k1[z]), ymm2, ymm3/m256/m32bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 65 /r VBLENDMPS zmm1 (k1[z]), zmm2, zmm3/m512/m32bcst</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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 an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

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

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

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

**VBLENDMPD (EVEX encoded versions)**

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

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

**VBLENDMPS (EVEX encoded versions)**

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

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no controlmask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN
          DEST[i+31:i] := SRC2[31:0]
        ELSE
          DEST[i+31:i] := SRC2[i+31:i]
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN DEST[i+31:i] := SRC1[i+31:i]
          ELSE ; zeroing-masking
            DEST[i+31:i] := 0
          FI;
      FI;
  END FOR
  DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VBLENDMPD __m512d _mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
VBLENDMPD __m256d _mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);
VBLENDMPD __m128d _mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);
VBLENDMPS __m512 _mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
VBLENDMPS __m256 _mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);
VBLENDMPS __m128 _mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions.”
## VBBROADCAST—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 VBBROADCASTSS xmm1, m32</td>
<td>A</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 VBBROADCASTSS ymm1, m32</td>
<td>A</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 VBBROADCASTSD ymm1, m64</td>
<td>A</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 VBBROADCASTF128 ymm1, m128</td>
<td>A</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>VEX.128.66.0F38.W0 18/r VBBROADCASTSS xmm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast the low single-precision floating-point element in the source operand to four locations in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 18 /r VBBROADCASTSS ymm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX</td>
<td>Broadcast low single-precision floating-point element in the source operand to eight locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 19 /r VBBROADCASTSD ymm1 (k1)[z], xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast 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 VBBROADCASTSD zmm1 (k1)[z], xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast 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 VBBROADCASTF32X2 ymm1 (k1)[z], xmm2/m64</td>
<td>C</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 VBBROADCASTF32X2 zmm1 (k1)[z], xmm2/m64</td>
<td>C</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 VBBROADCASTSS xmm1 (k1)[z], xmm2/m32</td>
<td>B</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 VBBROADCASTSS ymm1 (k1)[z], xmm2/m32</td>
<td>B</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 VBBROADCASTSS zmm1 (k1)[z], xmm2/m32</td>
<td>B</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 VBBROADCASTF32X4 ymm1 (k1)[z], m128</td>
<td>D</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 VBBROADCASTF32X4 zmm1 (k1)[z], m128</td>
<td>D</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 VBBROADCASTF64X2 ymm1 (k1)[z], m128</td>
<td>C</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>
</tbody>
</table>
VBROADCAST—Load with Broadcast Floating-Point Data

Description
VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

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

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

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/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 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. F32x4 and F64x4 are only supported as 512-bit wide versions.

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

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.
Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)
Operation

VBROADCASTSS (128 bit version VEX and legacy)

\[ \text{temp} := \text{SRC}[31:0] \]
\[ \text{DEST}[31:0] := \text{temp} \]
\[ \text{DEST}[63:32] := \text{temp} \]
\[ \text{DEST}[95:64] := \text{temp} \]
\[ \text{DEST}[127:96] := \text{temp} \]
\[ \text{DEST}[\text{MAXVL}-1:128] := 0 \]

VBROADCASTSS (VEX.256 encoded version)

\[ \text{temp} := \text{SRC}[31:0] \]
\[ \text{DEST}[31:0] := \text{temp} \]
\[ \text{DEST}[63:32] := \text{temp} \]
\[ \text{DEST}[95:64] := \text{temp} \]
\[ \text{DEST}[127:96] := \text{temp} \]
\[ \text{DEST}[159:128] := \text{temp} \]
\[ \text{DEST}[191:160] := \text{temp} \]
\[ \text{DEST}[223:192] := \text{temp} \]
\[ \text{DEST}[255:224] := \text{temp} \]
\[ \text{DEST}[\text{MAXVL}-1:256] := 0 \]

VBROADCASTSS (EVEX encoded versions)

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

Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)
**VBROADCASTSD (VEX.256 encoded version)**

\[
\begin{align*}
\text{temp} & := \text{SRC}[63:0] \\
\text{DEST}[63:0] & := \text{temp} \\
\text{DEST}[127:64] & := \text{temp} \\
\text{DEST}[191:128] & := \text{temp} \\
\text{DEST}[255:192] & := \text{temp} \\
\text{DEST}[\text{MAXVL}-1:256] & := 0
\end{align*}
\]

**VBROADCASTSD (EVEX encoded versions)**

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

\[
\begin{align*}
\text{FOR } j & := 0 \text{ TO } KL-1 \\
& j := j \times 64 \\
& \text{IF } \text{k1}[j] \text{ OR *no writemask*} \\
& \quad \text{THEN } \text{DEST}[i+63:i] := \text{SRC}[63:0] \\
& \quad \text{ELSE} \\
& \quad \quad \text{IF *merging-masking* ; merging-masking} \\
& \quad \quad \quad \text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged*} \\
& \quad \quad \quad \text{ELSE ; zeroing-masking} \\
& \quad \quad \quad \quad \text{DEST}[i+63:i] := 0 \\
& \quad \FI \\
& \FI
\end{align*}
\]

\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

**VBROADCASTF32x2 (EVEX encoded versions)**

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

\[
\begin{align*}
\text{FOR } j & := 0 \text{ TO } KL-1 \\
& j := j \times 32 \\
& n := (j \mod 2) \times 32 \\
& \text{IF } \text{k1}[j] \text{ OR *no writemask*} \\
& \quad \text{THEN } \text{DEST}[i+31:i] := \text{SRC}[n+31:n] \\
& \quad \text{ELSE} \\
& \quad \quad \text{IF *merging-masking* ; merging-masking} \\
& \quad \quad \quad \text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged*} \\
& \quad \quad \quad \text{ELSE ; zeroing-masking} \\
& \quad \quad \quad \quad \text{DEST}[i+31:i] := 0 \\
& \quad \FI \\
& \FI
\end{align*}
\]

\[
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

**VBROADCASTF128 (VEX.256 encoded version)**

\[
\begin{align*}
\text{temp} & := \text{SRC}[127:0] \\
\text{DEST}[127:0] & := \text{temp} \\
\text{DEST}[255:128] & := \text{temp} \\
\text{DEST}[\text{MAXVL}-1:256] & := 0
\end{align*}
\]
VBROADCASTF32X4 (EVEX encoded versions)
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  n := (j modulo 4) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[n+31:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VBROADCASTF64X2 (EVEX encoded versions)
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  n := (j modulo 2) * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := SRC[n+63:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
      FI
  FI;
ENDFOR;

VBROADCASTF32X8 (EVEX.U1.512 encoded version)
FOR j := 0 TO 15
  i := j * 32
  n := (j modulo 8) * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[n+31:n]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VBROADCASTF64X4 (EVEX.512 encoded version)

FOR j := 0 TO 7
  i := j * 64
  n := (j modulo 4) * 64
  IF k1\[j\] OR *no writemask*
      THEN DEST[i+63:i] := SRC[n+63:n]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTF32x2 __m512 _mm512broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m512 _mm512_maskbroadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskzbroadcast_f32x2(__mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m512 _mm512_maskbroadcast_f32x2(__m512 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskzbroadcast_f32x2(__mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m512 _mm512_maskbroadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskzbroadcast_f32x4(__mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m512 _mm512_maskbroadcast_f32x4(__m512 s, __mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskzbroadcast_f32x4(__mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512broadcast_f32x8(__m128 a);
VBROADCASTF32x8 __m512 _mm512_maskbroadcast_f32x8(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_maskzbroadcast_f32x8(__mmask16 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_broadcast_f32x8(__m128 a);
VBROADCASTF32x8 __m512 _mm512_maskbroadcast_f32x8(__m512 s, __mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_maskzbroadcast_f32x8(__mmask8 k, __m128 a);
VBROADCASTF64x2 __m512 _mm512broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m512 _mm512_maskbroadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512 _mm512_maskzbroadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x2 __m512 _mm512_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m512 _mm512_maskbroadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512 _mm512_maskzbroadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x2 __m512 _mm512_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m512 _mm512_maskbroadcast_f64x2(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x2 __m512 _mm512_maskzbroadcast_f64x2(__mmask8 k, __m256d a);
VBROADCASTF64x4 __m512 _mm512broadcast_f64x4(__m128d a);
VBROADCASTF64x4 __m512 _mm512_maskbroadcast_f64x4(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512 _mm512_maskzbroadcast_f64x4(__mmask8 k, __m256d a);
VBROADCASTF64x4 __m512 _mm512_broadcast_f64x4(__m128d a);
VBROADCASTF64x4 __m512 _mm512_maskbroadcast_f64x4(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512 _mm512_maskzbroadcast_f64x4(__mmask8 k, __m256d a);
VBROADCASTSD __m512d _mm512broadcast_sd(double *a);
VBROADCASTSD __m512d _mm512_maskbroadcastsd_pd(__m128d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512_maskzbroadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512_broadcastsd_pd(double *a);
VBROADCASTSS __m512d _mm512broadcastss_ps(__m128 a);
VBROADCASTSS __m512d _mm512_maskbroadcastss_ps(__m128 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512d _mm512_maskzbroadcastss_ps(__mmask16 k, __m128 a);
VBROADCASTSS __m512d _mm512_broadcastss_ps(__m128 a);
VBROADCASTSS __m512d _mm512_maskbroadcastss_ps(__m512b s, __mmask8 k, __m128 a);
VBROADCASTSS __m512d _mm512_maskzbroadcastss_ps(__mmask8 k, __m128 a);
VBROADCASTSS __m512d _mm512_broadcastss_ps(__m128 a);
VBROADCASTSS __m128_mm_broadcastss_ps(__m128 a);
VBROADCASTSS __m128_mm_mask_broadcastss_ps(__m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128_mm_maskz_broadcastss_ps(__mmask8 k, __m128 a);
VBROADCASTSS __m128_mm_broadcast_ss(float *a);
VBROADCASTSS __m256_mm256_broadcast_ss(float *a);
VBROADCASTF128 __m256_mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d_mm256_broadcast_pd(__m128d * a);

Exceptions
VEX-encoded instructions, see Table 2-23, "Type 6 Class Exception Conditions".
EVEX-encoded instructions, see Table 2-53, "Type E6 Class Exception Conditions".
Additionally:
#UD
  If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.
  If EVEX.L’L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.
  If EVEX.L’L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.
VCOMPRESSPD—Store Sparse Packed Double-Precision Floating-Point Values into Dense Memory

**Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</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)

\[
\begin{align*}
\text{SIZE} & := 64 \\
\text{k} & := 0 \\
\text{FOR } j & := 0 \text{ TO } \text{KL}-1 \\
\text{i} & := j \times 64 \\
\text{IF } k1[j] \text{ OR } \text{*no writemask*} \\
\text{THEN} \\
\text{DEST}[k+\text{SIZE}-1:k] & := \text{SRC}[i+63:i] \\
\text{k} & := k + \text{SIZE} \\
\text{FI;}
\end{align*}
\]

ENDFOR
VCOMPRESSPD (EVEX encoded versions) reg-reg form

(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN
    DEST[k+SIZE-1:k] := SRC[i+63:i]
    k := k + SIZE
  FI;
ENDFOR
IF *merging-masking*
  THEN *DEST[VL-1:k] remains unchanged*
  ELSE DEST[VL-1:k] := 0
FI
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCOMPRESSPD __m512d _mm512_mask_compress_pd(__m512d s, __mmask8 k, __m512d a);
VCOMPRESSPD __m512d _mm512_maskz_compress_pd(__mmask8 k, __m512d a);
VCOMPRESSPD void _mm512_mask_compressstoreu_pd(void * d, __mmask8 k, __m512d a);
VCOMPRESSPD __m256d _mm256_mask_compress_pd(__m256d s, __mmask8 k, __m256d a);
VCOMPRESSPD __m256d _mm256_maskz_compress_pd(__mmask8 k, __m256d a);
VCOMPRESSPD void _mm256_mask_compressstoreu_pd(void * d, __mmask8 k, __m256d a);
VCOMPRESSPD __m128d _mm_mask_compress_pd(__m128d s, __mmask8 k, __m128d a);
VCOMPRESSPD __m128d _mm_maskz_compress_pd(__mmask8 k, __m128d a);
VCOMPRESSPD void _mm_mask_compressstoreu_pd(void * d, __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions"; additionally:
#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>A</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>A</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>A</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>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</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 16 single-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

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

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

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

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

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

Operation
VCOMPRESSPS (EVEX encoded versions) store form
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 32
k := 0
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      DEST[k+SIZE-1:k] := SRC[i+31:i]
      k := k + SIZE
    FI;
ENDFOR;
VCOMPRESSPS (EVEX encoded versions) reg-reg form

(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE := 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[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCOMPRESSPS __m512 __m512_mask_compress_ps( __m512 s, __mmask16 k, __m512 a);
VCOMPRESSPS __m512 __m512_maskz_compress_ps( __mmask16 k, __m512 a);
VCOMPRESSPS void __m512_mask_compressstoreu_ps( void * d, __mmask16 k, __m512 a);
VCOMPRESSPS __m256 __m256_mask_compress_ps( __m256 s, __mmask8 k, __m256 a);
VCOMPRESSPS __m256 __m256_maskz_compress_ps( __mmask8 k, __m256 a);
VCOMPRESSPS void __m256_mask_compressstoreu_ps( void * d, __mmask8 k, __m256 a);
VCOMPRESSPS __m128 __m128_mask_compress_ps( __m128 s, __mmask8 k, __m128 a);
VCOMPRESSPS __m128 __m128_maskz_compress_ps( __mmask8 k, __m128 a);
VCOMPRESSPS void __m128_mask_compressstoreu_ps( void * d, __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E4.nb. in Table 2-49, “Type E4 Class Exception Conditions”; additionally:
#UD If EVEX.vvvv != 1111B.
VCVTNE2PS2BF16—Convert Two Packed Single Data to One Packed BF16 Data

### Description
Converts two SIMD registers of packed single data into a single register of packed BF16 data.

This instruction does not support memory fault suppression.

This instruction uses "Round to nearest (even)" rounding mode. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated. No floating-point exceptions are generated.

### Operation

#### VCVTNE2PS2BF16 dest, src1, src2

\[ \text{VL} = (128, 256, 512) \]

\[ \text{KL} = \text{VL}/16 \]

\[ \text{origdest} := \text{dest} \]

FOR \( i := 0 \) to \( \text{KL}-1 \):

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

IF \( i < \text{KL}/2 \):

IF src2 is memory and \( \text{evex.b} = 1 \):

\[ t := \text{src2.fp32[0]} \]

ELSE:

\[ t := \text{src2.fp32[i]} \]

ELSE:

\[ t := \text{src1.fp32[i-KL/2]} \]

// See VCVTNEPS2BF16 for definition of convert helper function

\[ \text{dest.word[i]} := \text{convert_fp32_to_bfloat16(t)} \]

ELSE IF *zeroing*:

\[ \text{dest.word[i]} := 0 \]

ELSE: // Merge masking, dest element unchanged

\[ \text{dest.word[i]} := \text{origdest.word[i]} \]

\[ \text{DEST[MAXVL-1:VL]} := 0 \]

---

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Opcode/Instruction

**EVEX.128.F2.0F38.W0 72 /r**

VCVTNE2PS2BF16 xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst

**EVEX.256.F2.0F38.W0 72 /r**

VCVTNE2PS2BF16 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst

**EVEX.512.F2.0F38.W0 72 /r**

VCVTNE2PS2BF16 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst

**AVX512_BF16**

Convert packed single data from xmm2 and xmm3/m128/m32bcst to packed BF16 data in xmm1 with writemask k1.

Convert packed single data from ymm2 and ymm3/m256/m32bcst to packed BF16 data in ymm1 with writemask k1.

Convert packed single data from zmm2 and zmm3/m512/m32bcst to packed BF16 data in zmm1 with writemask k1.
Intel C/C++ Compiler Intrinsic Equivalent
VCVTNE2PS2BF16 __m128bh _mm_cvtne2ps_pbh (__m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_mask_cvtne2ps_pbh (__m128bh, __mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m128bh _mm_maskz_cvtne2ps_pbh (__mmask8, __m128, __m128);
VCVTNE2PS2BF16 __m256bh _mm256_cvtne2ps_pbh (__m256, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_mask_cvtne2ps_pbh (__m256bh, __mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m256bh __mm256_maskz_cvtne2ps_pbh (__mmask16, __m256, __m256);
VCVTNE2PS2BF16 __m512bh __mm512_cvtne2ps_pbh (__m512, __m512);
VCVTNE2PS2BF16 __m512bh __mm512_mask_cvtne2ps_pbh (__m512bh, __mmask32, __m512, __m512);
VCVTNE2PS2BF16 __m512bh __mm512_maskz_cvtne2ps_pbh (__mmask32, __m512, __m512);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-50, "Type E4NF Class Exception Conditions".
VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data

**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>EVEX.128.F3.0F38.W0 72 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Convert packed single data from xmm2/m128 to packed BF16 data in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>VCVTNEPS2BF16 xmm1{k1}{z}, xmm2/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 72 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Convert packed single data from ymm2/m256 to packed BF16 data in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>VCVTNEPS2BF16 xmm1{k1}{z}, ymm2/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 72 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512_BF16</td>
<td>Convert packed single data from zmm2/m512 to packed BF16 data in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>VCVTNEPS2BF16 ymm1{k1}{z}, zmm2/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Converts one SIMD register of packed single data into a single register of packed BF16 data.

This instruction uses “Round to nearest (even)” rounding mode. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

As the instruction operand encoding table shows, the EVEX.vvvv field is not used for encoding an operand. EVEX.vvvv is reserved and must be 0b1111 otherwise instructions will #UD.

**Operation**

Define convert_fp32_to_bfloat16(x):

```
IF x is zero or denormal:
    dest[15] := x[31] // sign preserving zero (denormal go to zero)
    dest[14:0] := 0
ELSE IF x is infinity:
    dest[15:0] := x[31:16]
ELSE IF x is NAN:
    dest[15:0] := x[31:16] // truncate and set MSB of the mantissa to force QNAN
    dest[6] := 1
ELSE // normal number
    LSB := x[16]
    rounding_bias := 0x00007FFF + LSB
    temp[31:0] := x[31:0] + rounding_bias // integer add
    dest[15:0] := temp[31:16]
RETURN dest
```
VCVTNEPS2BF16 dest, src
VL = (128, 256, 512)
KL = VL/16

origdest := dest
FOR i := 0 to KL/2-1:
   IF k1[ i ] or *no writemask*:
      IF src is memory and evex.b == 1:
         t := src.fp32[0]
      ELSE:
         t := src.fp32[ i ]
      dest.word[ i ] := convert_fp32_to_bfloat16(t)
   ELSE IF *zeroing*:
      dest.word[ i ] := 0
   ELSE: // Merge masking, dest element unchanged
      dest.word[ i ] := origdest.word[ i ]
   DEST[MAXVL-1:VL/2] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTNEPS2BF16 __m128bh _mm_cvtneps_pbh (__m128);
VCVTNEPS2BF16 __m128bh _mm_mask_cvtneps_pbh (__m128bh, __mmask8, __m128);
VCVTNEPS2BF16 __m128bh _mm_maskz_cvtneps_pbh (__mmask8, __m128);
VCVTNEPS2BF16 __m128bh _mm256_cvtneps_pbh (__m256);
VCVTNEPS2BF16 __m128bh _mm256_mask_cvtneps_pbh (__m128bh, __mmask16, __m256);
VCVTNEPS2BF16 __m128bh _mm256_maskz_cvtneps_pbh (__mmask16, __m256);
VCVTNEPS2BF16 __m256bh _mm512_cvtneps_pbh (__m512);
VCVTNEPS2BF16 __m256bh _mm512_mask_cvtneps_pbh (__m256bh, __mmask16, __m512);
VCVTNEPS2BF16 __m256bh _mm512_maskz_cvtneps_pbh (__mmask16, __m512);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-49, “Type E4 Class Exception Conditions”.

VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data
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</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</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>VCVTPD2QQ xmm1 (k1)[z], xmm2/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 7B /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</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>VCVTPD2QQ ymm1 (k1)[z], ymm2/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 7B /r</td>
<td>A</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>
<tr>
<td>VCVTPD2QQ zmm1 (k1)[z], zmm2/m512/m64bcst{er}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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**

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

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

\[THEN\]

\[SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);\]

\[ELSE\]

\[SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);\]

\[FI;\]

\[FOR j := 0 TO KL-1\]

\[i := j * 64\]

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

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

\[ELSE\]

\[IF *merging-masking*\]

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

\[ELSE \quad ; zeroing-masking\]

\[DEST[i+63:i] := 0\]

\[FI\]

\[FI;\]

\[ENDFOR\]

\[DEST[MAXVL-1:VL] := 0\]

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

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

\[FOR j := 0 TO KL-1\]

\[i := j * 64\]

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

\[THEN\]

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

\[THEN\]

\[DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])\]

\[ELSE\]

\[DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])\]

\[FI;\]

\[ELSE\]

\[IF *merging-masking*\]

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

\[ELSE \quad ; zeroing-masking\]

\[DEST[i+63:i] := 0\]

\[FI\]

\[FI;\]

\[ENDFOR\]

\[DEST[MAXVL-1:VL] := 0\]
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2QQ __m512i _mm512_cvtpd_epi64( __m512d a);
VCVTPD2QQ __m512i _mm512_mask_cvtpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_maskz_cvtpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_cvtpd_roundpd_epi64( __m512d a, int r);
VCVTPD2QQ __m512i _mm512_mask_cvtpd_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m512i _mm512_maskz_cvtpd_roundpd_epi64( __mmask8 k, __m512d a);
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 Table 2-46, “Type E2 Class Exception Conditions”; additionally:
#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>A</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>A</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>A</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>Tuple Type</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>Full</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 (MAXVL-1:256) of the corresponding destination are zeroed.

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

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

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

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

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

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

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

VCVTPD2UDQ __m256i _mm512_cvtpd_epu32( __m512d a);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_maskz_cvtpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_cvt_roundpd_epu32( __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i _mm256_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm256_maskz_cvtpd_epu32( __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i _mm_maskz_cvtpd_epu32( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Table 2-46, ”Type E2 Class Exception Conditions”; additionally:
#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 /r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A/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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 /r VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A/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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 /r VCVTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst{er}</td>
<td>A/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>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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

VCVTPD2UQQ (EVEX encoded versions) when src operand is a register

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

IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

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

VCVTPD2UQQ (EVEX encoded versions) when src operand is a memory source

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

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
                FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
            FI
        FI;
    ELSE
        DEST[MAXVL-1:VL] := 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent
VCVTPD2UQQ __m512i _mm512_cvtpd_epu64(__m512d a);
VCVTPD2UQQ __m512i _mm512_mask_cvtpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_maskz_cvtpd_epu64(__mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_cvtpd_roundpd_epu64(__m512d a, int r);
VCVTPD2UQQ __m512i _mm512_mask_cvtpd_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_maskz_cvtpd_roundpd_epu64(__mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i _mm256_mask_cvtpd_epu64(__m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i _mm256_maskz_cvtpd_epu64(__mmask8 k, __m256d a);
VCVTPD2UQQ __m128i _mm_mask_cvtpd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i _mm_maskz_cvtpd_epu64(__mmask8 k, __m128d a);
VCVTPD2UQQ __m256i _mm256_cvtpd_epu64(__m256d src);
VCVTPD2UQQ __m128i _mm_cvtpd_epu64(__m128d src)

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”; additionally:
#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.128.66.0F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1 [k1][z], xmm2/m64</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1 [k1][z], xmm2/m128</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 13 /r VCVTPH2PS zmm1 [k1][z], ymm2/m256 (sae)</td>
<td>B</td>
<td>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>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Half Mem</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 (MAXVL-1:128) of the corresponding destination register are zeroed.

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

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

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

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

```
vCvt_h2s(SRC[15:0])
{
  RETURN Cvt_Half_Precision_To_Single_Precision(SRC[15:0]);
}
```

**VCVTPH2PS (EVEX encoded versions)**

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

FOR \(j := 0\) TO \(KL-1\)

\(i := j * 32\)
\(k := j * 16\)

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

THEN \(DEST[i+31:i] := vCvt_h2s(SRC[k+15:k])\)

ELSE

IF \(*merging-masking*\)

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

ELSE \(*zeroing-masking*\)

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

FI

FI

ENDFOR

\(DEST[\text{MAXVL-1:VL}] := 0\)

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

\(DEST[31:0] := vCvt_h2s(SRC[15:0]);\)
\(DEST[63:32] := vCvt_h2s(SRC[31:16]);\)
\(DEST[95:64] := vCvt_h2s(SRC[47:32]);\)
\(DEST[127:96] := vCvt_h2s(SRC[63:48]);\)
\(DEST[159:128] := vCvt_h2s(SRC[79:64]);\)
\(DEST[191:160] := vCvt_h2s(SRC[95:80]);\)
\(DEST[223:192] := vCvt_h2s(SRC[111:96]);\)
\(DEST[255:224] := vCvt_h2s(SRC[127:112]);\)
\(DEST[\text{MAXVL-1:256}] := 0\)

---

**Figure 5-6. VCVTPH2PS (128-bit Version)**

---
VCVTPH2PS (VEX.128 encoded version)

\[
\begin{align*}
\text{DEST}[31:0] & := \text{vCvt\_h2s}(\text{SRC1}[15:0]); \\
\text{DEST}[63:32] & := \text{vCvt\_h2s}(\text{SRC1}[31:16]); \\
\text{DEST}[95:64] & := \text{vCvt\_h2s}(\text{SRC1}[47:32]); \\
\text{DEST}[127:96] & := \text{vCvt\_h2s}(\text{SRC1}[63:48]); \\
\text{DEST}[\text{MAXVL-1}:128] & := 0
\end{align*}
\]

**Flags Affected**

None

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

VCVTPH2PS __m512 _mm512_cvtph_ps( __m256i a);
VCVTPH2PS __m512 _mm512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_maskz_cvtph_ps(__mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_cvt_roundph_ps( __m256i a, int sae);
VCVTPH2PS __m512 _mm512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m512 _mm512_maskz_cvt_roundph_ps(__mmask16 k, __m256i a, int sae);
VCVTPH2PS __m256 _mm256_cvtph_ps( __m256i a);
VCVTPH2PS __m256 _mm256_mask_cvtph_ps(__m256 s, __mmask8 k, __m128i a);
VCVTPH2PS __m256 _mm256_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_mask_cvtph_ps(__m128 s, __mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_cvtph_ps ( __m128i m1);
VCVTPH2PS __m256 _mm256_cvtph_ps ( __m128i m1)

**SIMD Floating-Point Exceptions**

Invalid

**Other Exceptions**

VEX-encoded instructions, see Table 2-26, “Type 11 Class Exception Conditions” (do not report #AC). EVEX-encoded instructions, see Table 2-60, “Type E11 Class Exception Conditions”.

Additionally:

- \#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 VCVTPS2PH xmm1/m64, xmm2, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 1D / r ib VCVTPS2PH xmm1/m128, ymm2, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 1D / r ib VCVTPS2PH xmm1/m64 [k1][z], xmm2, imm8</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 1D / r ib VCVTPS2PH xmm1/m128 [k1][z], ymm2, imm8</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1D / r ib VCVTPS2PH ymm1/m256 [k1][z], zmm2{sa}, imm8</td>
<td>B 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>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Half Mem</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 is listed in Table 5-3.

![Figure 5-7. VCVTPS2PH (128-bit Version)](image-url)
**VCTPS2PH**—Convert Single-Precision FP value to 16-bit FP value

**INSTRUCTION SET REFERENCE, V-Z**

Vol. 2C 5-41

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

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

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

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

**Operation**

\[
\text{vCvt_s2h}(\text{SRC}[31:0])
\]

\[
\begin{align*}
\text{IF } \text{Imm}[2] &= 0 \\
\text{THEN} \quad &\text{using Imm}[1:0] \text{ for rounding, see Table 5-3} \\
&\text{RETURN } \text{Cvt_Single_Precision_To_Half_Precision_FP}_\text{Imm} (\text{SRC}[31:0]); \\
\text{ELSE} \quad &\text{using MXCSR.RC for rounding} \\
&\text{RETURN } \text{Cvt_Single_Precision_To_Half_Precision_FP}_\text{Mxcsr} (\text{SRC}[31:0]); \\
\end{align*}
\]

\[
\text{FI}
\]

**VCVTPS2PH (EVEX encoded versions) when dest is a register**

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

\[
\text{FOR } j := 0 \text{ TO } KL-1 \\
i := j \times 16 \\
k := j \times 32 \\
IF k1[j] \text{ OR } *\text{no writemask}* \\
THEN \text{DEST}[i+15:i] := \\
\quad \text{vCvt_s2h}(\text{SRC}[k+31:k]) \\
ELSE \\
\quad IF *\text{merging-masking}* \\
\quad THEN *\text{DEST}[i+15:i] remains unchanged* \\
\quad ELSE \\
\quad \quad \text{DEST}[i+15:i] := 0 \\
\FI
\]

\[
\text{FI}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAXVL-1:VL/2}] := 0
\]

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>RC=00B</td>
<td>Round to nearest even</td>
<td>If Imm[2] = 0</td>
</tr>
<tr>
<td></td>
<td>RC=01B</td>
<td>Round down</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RC=10B</td>
<td>Round up</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RC=11B</td>
<td>Truncate</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>Ignored</td>
<td>Ignored by processor</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

- **VEX.vvvv** and **EVEX.vvvv** are reserved (must be 1111b).
- **EVEX encoded versions:** The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAXVL-1:256/128/64) of the corresponding destination register are zeroed.
VCVTPS2PH (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 16
  k := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := vCvt_s2h(SRC[k+31:k])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VCVTPS2PH (VEX.256 encoded version)
DEST[15:0] := vCvt_s2h(SRC1[31:0]);
DEST[31:16] := vCvt_s2h(SRC1[63:32]);
DEST[47:32] := vCvt_s2h(SRC1[95:64]);
DEST[63:48] := vCvt_s2h(SRC1[127:96]);
DEST[79:64] := vCvt_s2h(SRC1[159:128]);
DEST[95:80] := vCvt_s2h(SRC1[191:160]);
DEST[111:96] := vCvt_s2h(SRC1[223:192]);
DEST[127:112] := vCvt_s2h(SRC1[255:224]);
DEST[MAXVL-1:128] := 0

VCVTPS2PH (VEX.128 encoded version)
DEST[15:0] := vCvt_s2h(SRC1[31:0]);
DEST[31:16] := vCvt_s2h(SRC1[63:32]);
DEST[47:32] := vCvt_s2h(SRC1[95:64]);
DEST[63:48] := vCvt_s2h(SRC1[127:96]);
DEST[MAXVL-1:64] := 0

Flags Affected
None

Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2PH __m256i _mm512_cvtps_ph(__m512 a);
VCVTPS2PH __m256i _mm512_mask_cvtps_ph(__m256i s, __mmask16 k, __m512 a);
VCVTPS2PH __m256i _mm512_maskz_cvtps_ph(__mmask16 k, __m512 a);
VCVTPS2PH __m256i _mm512_cvt_roundps_ph(__m512 a, const int imm);
VCVTPS2PH __m256i _mm512_mask_cvt_roundps_ph(__m256i s, __mmask8 k, __m512 a);
VCVTPS2PH __m256i _mm512_maskz_cvt_roundps_ph(__mmask8 k, __m512 a);
SIMD Floating-Point Exceptions
Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);
Other Exceptions
VEX-encoded instructions, see Table 2-26, "Type 11 Class Exception Conditions" (do not report #AC);
EVEX-encoded instructions, see Table 2-60, "Type E11 Class Exception Conditions".
Additionally:
#UD If VEX.W=1.
#UD If VEX.vvv != 111B or EVEX.vvv != 111B.
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>A</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>A</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>A</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>Tuple Type</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>Full</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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;

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

DEST[MAXVL-1:VL] := 0

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

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

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

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2UDQ __m512i _mm512_cvtps_epu32( __m512 a);
VCVTPS2UDQ __m512i _mm512_mask_cvtps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_maskz_cvtps_epu32( __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_cvt_roundps_epu32( __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_mask_cvt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_maskz_cvt_roundps_epu32( __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i _mm256_cvtps_epu32( __m256d a);
VCVTPS2UDQ __m256i _mm256_mask_cvtps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i _mm256_maskz_cvtps_epu32( __mmask8 k, __m256 a);
VCVTPS2UDQ __m128i _mm_cvtps_epu32( __m128 a);
VCVTPS2UDQ __m128i _mm_mask_cvtps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i _mm_maskz_cvtps_epu32( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”; additionally:
#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>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 7B /r VCVTPS2QQ ymm1 {k1}(z), xmm2/m128/m32bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 7B /r VCVTPS2QQ zmm1 {k1}(z), ymm2/m256/m32bcst(er)</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Half</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-bit) 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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

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

ENDFOR

DEST[MAXVL-1:VL] := 0

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

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

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

ENDFOR

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2QQ __m512i _mm512_cvtps_epi64( __m512 a);
VCVTPS2QQ __m512i _mm512_mask_cvtps_epi64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_cvt_roundps_epi64( __m512 a, int r);
VCVTPS2QQ __m512i _mm512_mask_cvt_roundps_epi64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i _mm512_maskz_cvt_roundps_epi64( __mmask16 k, __m512 a, int r);

VCVTPS2QQ __m256i _mm256_cvtps_epi64( __m256 a);
VCVTPS2QQ __m256i _mm256_mask_cvtps_epi64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i _mm256_maskz_cvtps_epi64( __mmask8 k, __m256 a);
VCVTPS2QQ __m128i _mm_cvtps_epi64( __m128 a);
VCVTPS2QQ __m128i _mm_mask_cvtps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i _mm_maskz_cvtps_epi64( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”; additionally:
#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>A/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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A/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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 79 /r VCVTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}</td>
<td>A/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>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Half</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-bit) 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

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

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

\(\text{THEN}\)

\(\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);}\)

\(\text{ELSE}\)

\(\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);}\)

\(\text{FI;}\)

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

\(\text{i := j * 64}\)

\(\text{k := j * 32}\)

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

\(\text{THEN } \text{DEST}[i+63:i] := \text{Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])}\)

\(\text{ELSE}\)

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

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

\(\text{ELSE ; zeroing-masking}\)

\(\text{DEST}[i+63:i] := 0\)

\(\text{FI}\)

\(\text{FI}\)

\(\text{ENDFOR}\)

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)

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

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

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

\(\text{i := j * 64}\)

\(\text{k := j * 32}\)

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

\(\text{THEN}\)

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

\(\text{THEN}\)

\(\text{DEST}[i+63:i] := \text{Convert_Single_Precision_To_UQuadInteger(SRC[31:0])}\)

\(\text{ELSE}\)

\(\text{DEST}[i+63:i] := \text{Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])}\)

\(\text{FI;}\)

\(\text{ELSE}\)

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

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

\(\text{ELSE ; zeroing-masking}\)

\(\text{DEST}[i+63:i] := 0\)

\(\text{FI}\)

\(\text{FI}\)

\(\text{ENDFOR}\)

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)
Intel C/C++ Compiler Intrinsic Equivalent
VCVTPS2UQQ __m512i _mm512_cvtps_epu64(__m512 a);
VCVTPS2UQQ __m512i _mm512_mask_cvtps_epu64(__m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_maskz_cvtps_epu64(__mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_cvt_roundps_epu64(__m512 a, int r);
VCVTPS2UQQ __m512i _mm512_mask_cvt_roundps_epu64(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_maskz_cvt_roundps_epu64(__mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i _mm256_cvtps_epu64(__m256 a);
VCVTPS2UQQ __m256i _mm256_mask_cvtps_epu64(__m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i _mm256_maskz_cvtps_epu64(__mmask8 k, __m256 a);
VCVTPS2UQQ __m128i _mm_cvtps_epu64(__m128 a);
VCVTPS2UQQ __m128i _mm_mask_cvtps_epu64(__m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i _mm_maskz_cvtps_epu64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”; additionally:
#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.F3.0F.W1 E6 /r VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F.W1 E6 /r VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F.W1 E6 /r VCVTQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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 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

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

ELSE

\[
\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);}
\]

Fi;

FOR j := 0 TO KL-1

i := j * 64

IF k1[i] OR *no writemask*

THENDEST[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[MAXVL-1:VL] := 0
VCVTQQ2PD (EVEX encoded versions) when src operand is a memory source

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PD __m512d __m512_cvtepi64_pd( __m512i a);
VCVTQQ2PD __m512d __m512_mask_cvtepi64_pd( __m512d s, __mmask16 k, __m512i a);
VCVTQQ2PD __m512d __m512_maskz_cvtepi64_pd( __mmask16 k, __m512i a);
VCVTQQ2PD __m512d __m512_cvt_roundepi64_pd( __m512i a, int r);
VCVTQQ2PD __m512d __m512_mask_cvt_roundepi64_pd( __m512d s, __mmask8 k, __m512i a, int r);
VCVTQQ2PD __m512d __m512_maskz_cvt_roundepi64_pd( __mmask8 k, __m512i a);
VCVTQQ2PD __m256d __m256_cvtepi64_pd( __m256i a);
VCVTQQ2PD __m256d __m256_mask_cvtepi64_pd( __m256d s, __mmask8 k, __m256i a);
VCVTQQ2PD __m256d __m256_maskz_cvtepi64_pd( __mmask8 k, __m256i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions"; additionally:

#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>A</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>A</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>A</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>

**Description**

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

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

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

**Operation**

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

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PS __m256 _mm512_cvtepi64_ps( __m512i a);
VCVTQQ2PS __m256 _mm256_cvtepi64_ps( __m256i a);
VCVTQQ2PS __m128 _mm_cvtepi64_ps( __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”; additionally:

#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.LLIG.F2.0F.W0 79 /r VCVTSD2USI r32, xmm1/m64{er}</td>
<td>A</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.LLIG.F2.0F.W1 79 /r VCVTSD2USI r64, xmm1/m64{er}</td>
<td>A</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.</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>Tuple Type</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>Tuple1 Fixed</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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    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_cvtsroundsd_u32(__m128d, int r);
VCVTSD2USI unsigned __int64 _mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned __int64 _mm_cvt_roundsd_u64(__m128d, int r);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions".
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.LLIG.F3.0F.W0 79 /r VCVTSS2USI r32, xmm1/m32[er]</td>
<td>A</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.LLIG.F3.0F.W1 79 /r VCVTSS2USI r64, xmm1/m32[er]</td>
<td>A</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>Tuple Type</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>Tuple1 Fixed</td>
<td>ModRMreg (w)</td>
<td>ModRMrm/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

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

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

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

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

Operation

VCVTSS2USI (EVEX encoded version)

IF (SRC *is register*) AND (EVEX.b = 1)
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
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 __m128_mm_cvtss_u64(__m128 a);
VCVTSS2USI unsigned __int64 __m128_mm_cvt_roundss_u64(__m128 a, int r);
**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions”.
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 VCVTTTPD2QQ xmm1 {k1}[z]. xmm2/m128/m64bcst</td>
<td>A/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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 7A /r VCVTTTPD2QQ ymm1 {k1}[z]. ymm2/m256/m64bcst</td>
<td>A/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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 7A /r VCVTTTPD2QQ zmm1 {k1}[z]. zmm2/m512/m64bcst{sa}</td>
<td>A/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>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value \(2^w-1\), where \(w\) represents the number of bits in the destination format) is returned.

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

Operation

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

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

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0
VCVTTPD2QQ (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

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

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTPD2QQ __m512i _mm512_cvttpd_epi64( __m512d a);
VCVTTPD2QQ __m512i _mm512_mask_cvttpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_maskz_cvttpd_epi64( __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_cvtt_roundpd_epi64( __m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_mask_cvtt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_maskz_cvtt_roundpd_epi64( __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m256i _mm256_mask_cvttpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i _mm256_maskz_cvttpd_epi64( __mmask8 k, __m256d a);
VCVTTPD2QQ __m128i _mm_mask_cvttpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i _mm_maskz_cvttpd_epi64( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions"; additionally:
#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 /r VCVTTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</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 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.0F.W1 78 02 /r VCVTTPD2UDQ xmm1 {k1}[z], ymm2/m256/m64bcst</td>
<td>A</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 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.0F.W1 78 /r VCVTTPD2UDQ ymm1 {k1}[z], zmm2/m512/m64bcst{sae}</td>
<td>A</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 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value 2^w - 1 is returned, where w represents the number of bits in the destination format.

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

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

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

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

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

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

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

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

VCVTTPD2UDQ  __m256i  _mm512_cvttpd_epu32( __m512d a);
VCVTTPD2UDQ  __m256i  _mm512_mask_cvttpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ  __m256i  _mm512_maskz_cvttpd_epu32( __mmask8 k, __m512d a);
VCVTTPD2UDQ  __m256i  _mm512_cvtt_roundpd_epu32( __m512d a, int sae);
VCVTTPD2UDQ  __m256i  _mm512_mask_cvtt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ  __m256i  _mm512_maskz_cvtt_roundpd_epu32( __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ  __m128i  _mm256_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ  __m128i  _mm256_maskz_cvttpd_epu32( __mmask8 k, __m256d a);
VCVTTPD2UDQ  __m128i  _mm_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ  __m128i  _mm_maskz_cvttpd_epu32( __mmask8 k, __m128d a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions"; additionally:

#UD  If EVEX.vvvv != 1111B.
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 78 /r VCVTTPD2UQQ xmm1 {k1}[z], xmm2/m128/m64bcst</td>
<td>A</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 78 /r VCVTTPD2UQQ ymm1 {k1}[z], ymm2/m256/m64bcst</td>
<td>A</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 78 /r VCVTTPD2UQQ zmm1 {k1}[z], zmm2/m512/m64bcst{sae}</td>
<td>A</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>Tuple Type</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>Full</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 unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where $w$ represents the number of bits in the destination format.

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

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

**Operation**

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

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

FOR $j := 0$ TO $KL-1$

$i := j \times 64$

IF $k1[j]$ OR *no writemask*  
THEN $DEST[i+63:j] :=$  
Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate$(SRC[i+63:j])$

ELSE

IF *merging-masking*  
THEN *$DEST[i+63:j]$ remains unchanged*  
ELSE  
$DEST[i+63:j] := 0$

FI

ENDFOR

$DEST[MAXVL-1:VL] := 0$
VCVTPD2UQQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b == 1)
    THEN
      DEST[i+63:i] :=
      Convert_Double_Precision_Floating_Point_To_UQuadInteger_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[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2UQQ _mm<size>[_mask[z]]_cvtt[_round]pd_epu64
VCVTPD2UQQ _mm512i _mm512_cvtpd_epu64(_mm512d a);
VCVTPD2UQQ _mm512i _mm512_mask_cvtpd_epu64(_mm512i s, _mmask8 k, _mm512d a);
VCVTPD2UQQ _mm512i _mm512_maskz_cvtpd_epu64(_mmask8 k, _mm512d a);
VCVTPD2UQQ _mm512i _mm512_cvtt_roundpd_epu64(_mm512d a, int sae);
VCVTPD2UQQ _mm512i _mm512_mask_cvtt_roundpd_epu64(_mm512i s, _mmask8 k, _mm512d a, int sae);
VCVTPD2UQQ _mm512i _mm512_maskz_cvtt_roundpd_epu64(_mmask8 k, _mm512d a, int sae);
VCVTPD2UQQ _mm256i _mm256_mask_cvtt_roundpd_epu64(_mm256i s, _mmask8 k, _mm256d a);
VCVTPD2UQQ _mm256i _mm256_maskz_cvtt_roundpd_epu64(_mmask8 k, _mm256d a);
VCVTPD2UQQ _mm128i _mm_mask_cvtt_roundpd_epu64(_mm128i s, _mmask8 k, _mm128d a);
VCVTPD2UQQ _mm128i _mm_maskz_cvtt_roundpd_epu64(_mmask8 k, _mm128d a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”; additionally:
#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>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.0F.W0 78 /r VCVTTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.0F.W0 78 /r VCVTTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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, a truncated (round toward zero) value is returned. 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 \(KL-1\)

\(i := j \times 32\)

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

ELSE  
\(DEST[i+31:i] := 0\)

\(FI\)

ENDFOR

\(DEST[MAXVL-1:VL] := 0\)
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[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2UDQ __m512i _mm512_cvttps_epu32( __m512 a);
VCVTTPS2UDQ __m512i _mm512_mask_cvttps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_maskz_cvttps_epu32( __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_cvtt_roundps_epu32( __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_mask_cvtt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_maskz_cvtt_roundps_epu32( __mmask16 k, __m512 a);
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 Table 2-46, “Type E2 Class Exception Conditions”; additionally:
#UD If EVEX.vvvv != 1111B.
VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

## 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>Instruction Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F:w0 7A/r</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td></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</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td></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</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td></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>Tuple Type</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>Half</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 eight signed quadword integers in the destination operand.

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

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

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

## Operation

**VCVTTPS2QQ (EVEX encoded versions) when src operand is a register**

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

FOR \(j := 0\) TO KL-1

\(i := j \times 64\)

\(k := j \times 32\)

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

THEN \(\text{DEST}[i+63:i] := \text{Convert Single Precision To QuadInteger Truncate} (\text{SRC}[k+31:k])\)

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

\(\text{DEST}[i+63:i] := 0\)

FI

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)
VCVTTPS2QQ (EVEX encoded versions) when src operand is a memory source

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

For \(j := 0\) to \(KL-1\)

\(i := j \times 64\)
\(k := j \times 32\)

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

Then

If (EVEX.b == 1)

Then

\(\text{DEST}[i+63:i] :=\)

Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])

Else

\(\text{DEST}[i+63:i] :=\)

Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])

Else

If "merging-masking"

Then "DEST[i+63:i] remains unchanged"

Else "zeroing-masking"

\(\text{DEST}[i+63:i] := 0\)

Else

\(\text{DEST}[MAXVL-1:VL] := 0\)

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{VCVTTPS2QQ } & \_m512i \_mm512\text{-cvttps_epi64( } \_m256 \text{ a);} \\
\text{VCVTTPS2QQ } & \_m512i \_mm512\text{-mask_cvttps_epi64( } \_m512i \_s, \_mmask16 \_k, \_m256 \_a); } \\
\text{VCVTTPS2QQ } & \_m512i \_mm512\text{-maskz_cvttps_epi64( } \_mmask16 \_k, \_m256 \_a); } \\
\text{VCVTTPS2QQ } & \_m512i \_mm512\text{-cvtt_roundps_epi64( } \_m256 \_a, \text{int sae);} \\
\text{VCVTTPS2QQ } & \_m512i \_mm512\text{-mask_cvtt_roundps_epi64( } \_m512i \_s, \_mmask16 \_k, \_m256 \_a, \text{int sae);} \\
\text{VCVTTPS2QQ } & \_m256i \_mm256\text{-cvttps_epi64( } \_m256 \_a, \_s, \_mmask8 \_k, \_m128 \_a); } \\
\text{VCVTTPS2QQ } & \_m256i \_mm256\text{-mask_cvttps_epi64( } \_mmask8 \_k, \_m128 \_a); } \\
\text{VCVTTPS2QQ } & \_m128i \_mm256\text{-mask_cvttps_epi64( } \_m128i \_s, \_mmask8 \_k, \_m128 \_a); } \\
\text{VCVTTPS2QQ } & \_m128i \_mm256\text{-maskz_cvttps_epi64( } \_mmask8 \_k, \_m128 \_a); }
\end{align*}
\]

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

Evex-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”; additionally:

\#UD If EVEX.vvvv != 1111B.
VCVTPPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer 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.128.66.0F:w0 78 /r VCVTTPS2UQQ xmm1 {k1}[z], xmm2/m64/m32bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F:w0 78 /r VCVTTPS2UQQ ymm1 [k1][z], xmm2/m128/m32bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F:w0 78 /r VCVTTPS2UQQ zmm1 [k1][z], xmm2/m256/m32bcst{sae}</td>
<td>A V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from xmm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Half</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, a truncated (round toward zero) value is returned. 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

**VCVTPPS2UQQ (EVEX encoded versions) when src operand is a register**

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

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[{j}] OR *no writemask*
    THEN DEST[i+63:i] := Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VCVTPPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values
VCVTTPS2UQQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

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

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTPS2UQQ _mm<size>_[_mask[z]]_cvtt[_round]ps_epu64
VCVTTPS2UQQ _m512i _mm512_cvttps_epu64(__m256 a);
VCVTTPS2UQQ _m512i _mm512_mask_cvttps_epu64(__m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ _m512i _mm512_maskz_cvttps_epu64(__mmask16 k, __m256 a);
VCVTTPS2UQQ _m512i _mm512_cvtt_roundps_epu64(__m256 a, int sae);
VCVTTPS2UQQ _m512i _mm512_mask_cvtt_roundps_epu64(__m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ _m512i _mm512_maskz_cvtt_roundps_epu64(__mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ _m256i _mm256_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 Table 2-47, "Type E3 Class Exception Conditions"; additionally:
#UD If EVEX.vvvv != 1111B.
VCVTTSD2USI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Unsigned 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.VLIG.F2.0F.W0 78 /r</td>
<td>A</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{sae}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.VLIG.F2.0F.W1 78 /r</td>
<td>A</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{sae}</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>Tuple Type</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>Tuple1 Fixed</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, a truncated (round toward zero) value is returned. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value \(2^w - 1\) is returned, where \(w\) represents the number of bits in the destination format.

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

Operation

**VCVTTSD2USI (EVEX encoded version)**

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2USI unsigned int _mm_cvtt_s32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvtt_s64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundsd_u64(__m128d, int sae);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions".
VCVTTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned 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.LLIG.F3.0F.W0 7B/r VCVTTSS2USI r32, xmm1/m32{saе}</td>
<td>A</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>EVEX.LLIG.F3.0F.W1 7B/r VCVTTSS2USI r64, xmm1/m32{saе}</td>
<td>A</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 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>Tuple Type</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>Tuple1 Fixed</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, a truncated (round toward zero) value is returned. 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
VCVTTSS2USI (EVEX encoded version)
IF 64-bit Mode and OperandSize = 64
THEN
DEST[63:0] := Convert_Single_Precision_Floating_Point_To_UInteger_Truncate{SRC[31:0]};
ELSE
DEST[31:0] := Convert_Single_Precision_Floating_Point_To_UInteger_Truncate{SRC[31:0]};
FI;

Intel C/C++ Compiler Intrinsic Equivalent
VCVTTSS2USI unsigned int _mm_cvttss_u32(__m128 a);
VCVTTSS2USI unsigned int _mm_cvtt_roundss_u32(__m128 a, int sae);
VCVTTSS2USI unsigned __int64 _mm_cvttss_u64(__m128 a);
VCVTTSS2USI unsigned __int64 _mm_cvtt_roundss_u64(__m128 a, int sae);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions".
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>
</table>
| EVEX.128.F3.0F.W0 7A /r
VCVTUDQ2PD xmm1 {k1}[z], xmm2/m64/m32bcst | A       | V/V                   | AVX512VL AVX512F | Convert two packed unsigned doubleword integers from xmm2/m64/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1. |
| EVEX.256.F3.0F.W0 7A /r
VCVTUDQ2PD ymm1 {k1}[z], xmm2/m128/m32bcst | A       | V/V                   | AVX512VL AVX512F | Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1. |
| EVEX.512.F3.0F.W0 7A /r
VCVTUDQ2PD zmm1 {k1}[z], ymm2/m256/m32bcst | A       | V/V                   | AVX512F          | Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1. |

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Half</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 \(KL-1\)

\(i := j \times 64\)

\(k := j \times 32\)

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

THEN \(\text{DEST}[i+63:] := \text{Convert_UIInteger_To_Double_Precision_Floating_Point}(\text{SRC}[k+31:k])\)

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

\(\text{DEST}[i+63:] := 0\)

FI

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)
VCVTUDQ2PD (EVEX encoded versions) when src operand is a memory source

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

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+63:i] := Convert_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[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Table 2-51, "Type E5 Class Exception Conditions"; additionally:

#UD If EVEX.vvvv != 1111B.
VCVTUDQ2PS—Convert Packed Unsigned Doubleword 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.F2.0F:W0 7A /r</td>
<td>A</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.256.F2.0F:W0 7A /r</td>
<td>A</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.512.F2.0F:W0 7A /r</td>
<td>A</td>
<td>V/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>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

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

FI;
ENDFOR

DEST[MAXVL-1:VL] := 0
VCVTUDQ2PS (EVEX encoded version) when src operand is a memory source

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

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] :=
                    Convert_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[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PS __m512 _mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m512 __mm512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m526 __mm526_cvtepu32_ps( __m526i a);
VCVTUDQ2PS __m512 __mm526_mask_cvtepu32_ps( __m512 s, __mmask8 k, __m526i a);
VCVTUDQ2PS __m512 __mm512_maskz_cvtepu32_ps( __mmask8 k, __m512i 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 Table 2-46, "Type E2 Class Exception Conditions"; additionally:

#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 VCVTUQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A</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 VCVTUQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A</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 VCVTUQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst(er)</td>
<td>A</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>Tuple Type</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0/1</td>
<td>Full</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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j * 64

IF k1[j] OR *no writemask*

THEN DEST[i+63:i] :=

Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0
VCVTUQQ2PD (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

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

Intel C/C++ Compiler Intrinsic Equivalent
VCVTUQQ2PD __m512d _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d _mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_mask_cvt_roundepu64_ps( __m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d _mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m128d _mm128_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d _mm128_mask_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d _mm128_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions"; additionally:
#UD If EVEX.vvvv != 1111B.
VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

Opcode/Instruction | Op / En | 64/32 bit Mode Support | CPUID Feature Flag | Description
--- | --- | --- | --- | ---
EVEX.128.F2.0F.W1 7A / r | A | V/V | AVX512VL AVX512DQ | Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single-precision floating-point values in zmm1 with writemask k1.
EVEX.256.F2.0F.W1 7A / r | A | V/V | AVX512VL AVX512DQ | Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.F2.0F.W1 7A / r | A | V/V | AVX512DQ | Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single-precision floating-point values in zmm1 with writemask k1.

Instruction Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE

SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

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

ENDFOR

DEST[MAXVL-1:VL/2] := 0
VCVTUQQ2PS (EVEX encoded version) when src operand is a memory source

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PS __m256 _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PS __m256 _mm512_mask_cvtepu64_ps( __m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PS __m256 _mm512_mask_cvt_roundepu64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PS __m256 _mm256_mask_cvtepu64_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m256 _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PS __m256 _mm256_cvt_roundepu64_ps( __m256i a, int r);
VCVTUQQ2PS __m256 _mm256_mask_cvt_roundepu64_ps( __m256 s, __mmask8 k, __m256i a, int r);
VCVTUQQ2PS __m128 _mm512_cvtepu64_ps( __m128i a);
VCVTUQQ2PS __m128 _mm512_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm512_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm512_cvt_roundepu64_ps( __m128i a, int r);
VCVTUQQ2PS __m128 _mm512_mask_cvt_roundepu64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 _mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”; additionally:
#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.LLIG.F2.0F.W0 7B /r</td>
<td>A</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.LLIG.F2.0F.W1 7B /r</td>
<td>A</td>
<td>V/N.E.¹</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 instruction behaves as if the W0 version is used.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</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 (MAXVL-1:128) of the destination register are zeroed.

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

Operation

VCVTUSI2SD (EVEX encoded version)

IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF 64-Bit Mode And OperandSize = 64
    THEN
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[63:0]);
    ELSE
    DEST[63:0] := Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[31:0]);
    FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTUSI2SD __m128d __mm_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 Table 2-48, “Type E3NF Class Exception Conditions” if W1, else see Table 2-59, “Type E10NF Class Exception Conditions”.
VCVTUSI2SS—Convert Unsigned Integer to Scalar Single-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.LLIG.F3.0F.W0 7B /r VCVTUSI2SS xmm1, xmm2, r/m32{er}</td>
<td>A</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.LLIG.F3.0F.W1 7B /r VCVTUSI2SS xmm1, xmm2, r/m64{er}</td>
<td>A</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.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

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

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

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

Operation
VCVTUSI2SS (EVEX encoded version)
IF (SRC2 *is register*) AND (EVEX.b = 1)
THEN
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
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[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VCVTUSI2SS __m128_mm_cvtu32_ss (__m128 s, unsigned a);
VCVTUSI2SS __m128_mm_cvtu_roundu32_ss(__m128 s, unsigned a, int r);
VCVTUSI2SS __m128_mm_cvtu64_ss (__m128 s, unsigned __int64 a);
VCVTUSI2SS __m128_mm_cvtu_roundu64_ss( __m128 s, unsigned __int64 a, int r);
SIMD Floating-Point Exceptions
Precision

Other Exceptions
See Table 2-48, “Type E3NF Class Exception Conditions”.
VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

### Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes in 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

\[
\begin{align*}
\text{TMP1}[I+31:I] &:= \text{select} \ (\text{SRC2}[I+127:I], \text{imm8}[1:0]) \\
\text{TMP1}[I+63:I+32] &:= \text{select} \ (\text{SRC2}[I+127:I], \text{imm8}[3:2]) \\
\text{TMP1}[I+95:I+64] &:= \text{select} \ (\text{SRC2}[I+127:I], \text{imm8}[5:4]) \\
\text{TMP1}[I+127:I+96] &:= \text{select} \ (\text{SRC2}[I+127:I], \text{imm8}[7:6])
\end{align*}
\]

END FOR

SAD of quadruplets:

FOR \( I = 0 \) to \( VL \) step 64

\[
\begin{align*}
\text{TMP_DEST}[I+15:I] &:= \text{ABS}(\text{SRC1}[I+7:I] - \text{TMP1}[I+7:I]) + \\
&\quad \text{ABS}(\text{SRC1}[I+15:I+8] - \text{TMP1}[I+15:I+8]) + \\
&\quad \text{ABS}(\text{SRC1}[I+23:I+16] - \text{TMP1}[I+23:I+16]) + \\
&\quad \text{ABS}(\text{SRC1}[I+31:I+24] - \text{TMP1}[I+31:I+24])
\end{align*}
\]

\[
\begin{align*}
\text{TMP_DEST}[I+31:I+16] &:= \text{ABS}(\text{SRC1}[I+7:I] - \text{TMP1}[I+15:I+8]) + \\
&\quad \text{ABS}(\text{SRC1}[I+15:I+16] - \text{TMP1}[I+15:I+16]) + \\
&\quad \text{ABS}(\text{SRC1}[I+23:I+16] - \text{TMP1}[I+23:I+16]) + \\
&\quad \text{ABS}(\text{SRC1}[I+31:I+24] - \text{TMP1}[I+31:I+24])
\end{align*}
\]

\[
\begin{align*}
\text{TMP_DEST}[I+47:I+32] &:= \text{ABS}(\text{SRC1}[I+39:I+32] - \text{TMP1}[I+23:I+16]) + \\
&\quad \text{ABS}(\text{SRC1}[I+47:I+40] - \text{TMP1}[I+31:I+24]) + \\
&\quad \text{ABS}(\text{SRC1}[I+55:I+48] - \text{TMP1}[I+39:I+32]) + \\
&\quad \text{ABS}(\text{SRC1}[I+63:I+56] - \text{TMP1}[I+47:I+40])
\end{align*}
\]

\[
\begin{align*}
\text{TMP_DEST}[I+63:I+48] &:= \text{ABS}(\text{SRC1}[I+39:I+32] - \text{TMP1}[I+31:I+24]) + \\
&\quad \text{ABS}(\text{SRC1}[I+47:I+40] - \text{TMP1}[I+39:I+32]) + \\
&\quad \text{ABS}(\text{SRC1}[I+55:I+48] - \text{TMP1}[I+47:I+40]) + \\
&\quad \text{ABS}(\text{SRC1}[I+63:I+56] - \text{TMP1}[I+55:I+48])
\end{align*}
\]

ENDFOR

FOR \( j := 0 \) TO \( KL-1 \)

\[
i := j \times 16
\]

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

THEN \( \text{DEST}[i+15:i] := \text{TMP_DEST}[i+15:i] \)
ELSE

IF *merging-masking* ; merging-masking

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

\( \text{DEST}[i+15:i] := 0 \)

FI

FI;

ENDFOR

\( \text{DEST}[\text{MAXVL}-1:VL] := 0 \)
Intel C/C++ Compiler Intrinsic Equivalent
VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b int imm8);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b int imm8);
VDBPSADBW __m128i _mm_dbsad_epu8(__m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b int imm8);
VDBPSADBW __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b int imm8);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions".
**VDPBF16PS—Dot Product of BF16 Pairs Accumulated into Packed Single Precision**

<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 52 /r VDPBF16PS xmm1(k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512 BF16</td>
<td>Multiply BF16 pairs from xmm2 and xmm3/m128, and accumulate the resulting packed single precision results in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 52 /r VDPBF16PS ymm1(k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512 BF16</td>
<td>Multiply BF16 pairs from ymm2 and ymm3/m256, and accumulate the resulting packed single precision results in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 52 /r VDPBF16PS zmm1(k1){z}, zmm2, zmm3/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512 BF16</td>
<td>Multiply BF16 pairs from zmm2 and zmm3/m512, and accumulate the resulting packed single precision results in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Full</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 dot-product of two BF16 pairs and accumulates into a packed single precision register.

“Round to nearest even” rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

NaN propagation priorities are described in Table 5-1.

**Table 5-1. NaN Propagation Priorities**

<table>
<thead>
<tr>
<th>NaN Priority</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>src1 low is NaN</td>
<td>Lower part has priority over upper part, i.e., it overrides the upper part.</td>
</tr>
<tr>
<td>2</td>
<td>src2 low is NaN</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>src1 high is NaN</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>src2 high is NaN</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>srcdest is NaN</td>
<td>Dest is propagated if no NaN is encountered by src2.</td>
</tr>
</tbody>
</table>

**Operation**

Define make_fp32(x):

```
// The x parameter is bfloat16. Pack it in to upper 16b of a dword. The bit pattern is a legal fp32 value. Return that bit pattern.
dword := 0
dword[31:16] := x
RETURN dword
```
VDPBF16PS srcdest, src1, src2
VL = (128, 256, 512)
KL = VL/32

origdest := srcdest
FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF src2 is memory and evex.b == 1:
            t := src2.dword[0]
        ELSE:
            t := src2.dword[i]
        // FP32 FMA with daz in, ftz out and RNE rounding. MXCSR neither consulted nor updated.
        srcdest.fp32[i] += make_fp32(src1.bfloat16[2*i+1]) * make_fp32(t.bfloat1)
        srcdest.fp32[i] += make_fp32(src1.bfloat16[2*i+0]) * make_fp32(t.bfloat0)
    ELSE IF *zeroing*:
        srcdest.dword[i] := 0
    ELSE: // merge masking, dest element unchanged
        srcdest.dword[i] := origdest.dword[i]
srcdest[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VDPBF16PS __m128 _mm_dpbf16_ps(__m128, __m128bh, __m128bh);
VDPBF16PS __m128 __mm_mask_dpbf16_ps(__m128, __mmask8, __m128bh, __m128bh);
VDPBF16PS __m128 __mm_maskz_dpbf16_ps(__mmask8, __m128, __m128bh, __m128bh);
VDPBF16PS __m256 __mm256_dpbf16_ps(__m256, __m256bh, __m256bh);
VDPBF16PS __m256 __mm256_mask_dpbf16_ps(__m256, __mmask8, __m256bh, __m256bh);
VDPBF16PS __m256 __mm256_maskz_dpbf16_ps(__mmask8, __m256, __m256bh, __m256bh);
VDPBF16PS __m512 __mm512_dpbf16_ps(__m512, __m512bh, __m512bh);
VDPBF16PS __m512 __mm512_mask_dpbf16_ps(__m512, __mmask16, __m512bh, __m512bh);
VDPBF16PS __m512 __mm512_maskz_dpbf16_ps(__mmask16, __m512, __m512bh, __m512bh);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-49, “Type E4 Class Exception Conditions”.

VDPBF16PS—Dot Product of BF16 Pairs Accumulated into Packed Single Precision
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>A</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>A</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>A</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>Tuple Type</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>Tuple1 Scalar</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[MAXVL-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_mask_expandloadu_pd( __m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d _mm512_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_mask_expand_pd( __m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_maskz_expand_pd( __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_mask_expandloadu_pd( __m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m128d _mm_mask_expand_pd( __m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_maskz_expand_pd( __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_maskz_expandloadu_pd( __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 in Table 2-49, “Type E4 Class Exception Conditions”; additionally:

#UD If EVEX.vvvv != 1111B.
VEXPANDPS—Load Sparse Packed Single-Precision Floating-Point Values from Dense Memory

![Table of Opcode Instructions]

<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 VEXPANDPS xmm1 {k1}[z], xmm2/m128</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}[z], ymm2/m256</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}[z], zmm2/m512</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Expand packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

![Instruction Operand Encoding]

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</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)**

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

\(k := 0\)

\(\mathrm{FOR} \ j := 0 \ \mathrm{TO} \ KL-1\)

\(i := j \times 32\)

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

\(\text{THEN}\)

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

\(k := k + 32\)

\(\text{ELSE}\)

\(\text{IF } \ast \text{merging-mask}\ast \)

\(\text{THEN } \ast \text{DEST}[i+31:i] \text{ remains unchanged}\ast \)

\(\text{ELSE } \ast \text{zeroing-mask}\ast \)

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

\(\text{FI}\)

\(\text{ENDFOR}\)

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

VEXPANDPS __m512 _mm512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPS __m512 _mm512_maskz_loadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPD __m256 _mm256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_maskz_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 _mm256_maskz_loadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 _mm_mask_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_maskz_expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_maskz_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 _mm_maskz_loadu_ps( __mmask8 k, void * a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”; additionally:

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

### Opcode

<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;
            ELSE
                ZF := 0;
        ELSE
            ZF := 0;

### Opcode

<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>
Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

Protected Mode Exceptions

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

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

- **#SS(0)** If a memory operand effective address is outside the SS segment limit.

- **#PF(fault-code)** If a page fault occurs.

- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

- **#UD** If the LOCK prefix is used.

Real-Address Mode Exceptions

- **#UD** The VERR and VERW instructions are not recognized in real-address mode.
  - If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

- **#UD** The VERR and VERW instructions are not recognized in virtual-8086 mode.
  - If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used.
VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values

<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 VEXTRACTF128 xmm1/m128, ymm2, imm8</td>
<td>A</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,w1 19 / r ib VEXTRACTF32x4 xmm1/m128-k1]z, ymm2, imm8</td>
<td>C</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,w1 19 / r ib VEXTRACTF64x2 xmm1/m128-k1]z, zmm2, imm8</td>
<td>C</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 VEXTRACTF64X2 xmm1/m128-k1]z, ymm2, imm8</td>
<td>B</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,w1 19 / r ib VEXTRACTF64X2 xmm1/m128-k1]z, zmm2, imm8</td>
<td>B</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,w1 19 / r ib VEXTRACTF64X4 ymm1/m256-k1]z, zmm2, imm8</td>
<td>D</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,w1 19 / r ib VEXTRACTF64x4 ymm1/m256-k1]z, zmm2, imm8</td>
<td>C</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>Tuple Type</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>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</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 an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor The destination may be either a vector register or a 256-bit memory location.

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

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

The high 6 bits of the immediate are ignored.

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

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]
  ESAC.
FI;
IF VL = 512
  CASE (imm8[1:0]) OF
    00: TMP_DEST[127:0] := SRC1[127:0]
  ESAC.
FI;
FOR j := 0 TO 3
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:128] := 0

VEXTRACTF32x4 (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]
  ESAC.
FI;
IF VL = 512
  CASE (imm8[1:0]) OF
    00: TMP_DEST[127:0] := SRC1[127:0]
  ESAC.
FI;
FOR j := 0 TO 3
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
  FI;
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]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
    ESAC.
FI;
FOR j := 0 TO 1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:128] := 0

VEXTRACTF64x2 (EVEX encoded versions) when destination is memory
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
    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[MAXVL-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
    IF 
    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[Δ+63]: = 0

FI;
ENDFOR
DEST[MAXVL-1:256]: = 0

**VEXTRACTF64x4 (EVEX.512 encoded version) when destination is memory**

CASE (imm8[0]) OF

0: TMP_DEST[255:0] := SRC1[255:0]
1: TMP_DEST[255:0] := SRC1[511:256]

ESAC.

FOR j := 0 TO 3
i := j * 64
IF k1[[]] OR "no writemask"

THEN DEST[i+63]: = TMP_DEST[i+63]
ELSE ; merging-masking

*DEST[i+63] remains unchanged*

FI;
ENDFOR

**VEXTRACTF128 (memory destination form)**

CASE (imm8[0]) OF

0: DEST[127:0] := SRC1[127:0]

ESAC.

**VEXTRACTF128 (register destination form)**

CASE (imm8[0]) OF

0: DEST[127:0] := SRC1[127:0]

ESAC.

DEST[MAXVL-1:128] := 0

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

VEXTRACTF32x4 __m128 __mm512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128 __mm512_mask_extractf32x4_ps(__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);

VEXTRACTF128 __m128 __mm256_extractf128_ps (__m256 a, int offset);
VEXTRACTF128 __m128d_mm256_extractf128_pd (__m256d a, int offset);
VEXTRACTF128 __m128i_mm256_extractf128_si256(__m256i a, int offset);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-54, “Type E6NF Class Exception Conditions”.
Additionally:

- #UD IF VEX.L = 0.
- #UD IF VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

### Opcode/Instruction

<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 VEXTRACTI128</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.w0 39 /r ib VEXTRACTI32x4</td>
<td>C 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.w0 39 /r ib VEXTRACTI32x4</td>
<td>C 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>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F3A.w1 39 /r ib VEXTRACTI64x2</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.w1 39 /r ib VEXTRACTI64x2</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.w0 3B /r ib VEXTRACTI64x4</td>
<td>D 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.w1 3B /r ib VEXTRACTI64x4</td>
<td>C 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>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</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 a 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]
   ESAC.
   FI;

IF VL = 512
   CASE (imm8[1:0]) OF
      00: TMP_DEST[127:0] := SRC1[127:0]
   ESAC.
   FI;

FOR j := 0 TO 3
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := TMP_DEST[i+31:i]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
         DEST[i+31:i] := 0
      FI
   FI
ENDFOR
DEST[MAXVL-1:128] := 0

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

VL = 256, 512

IF VL = 256
   CASE (imm8[0]) OF
      0: TMP_DEST[127:0] := SRC1[127:0]
   ESAC.
   FI;

IF VL = 512
   CASE (imm8[1:0]) OF
      00: TMP_DEST[127:0] := SRC1[127:0]
   ESAC.

5-106 Vol. 2C
FI;

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

VEXTRACTI64x2 (EVEX encoded versions) when destination is a register
VL = 256, 512
IF VL = 256
  CASE (imm8[0]) OF
    0: TMP_DEST[127:0] := SRC1[127:0]
  ESAC.
  FI;
IF VL = 512
  CASE (imm8[1:0]) OF
    00: TMP_DEST[127:0] := SRC1[127:0]
  ESAC.
  FI;
FOR j := 0 TO 1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:128] := 0
VEXTRACTI64x2 (EVEX encoded versions) when destination is memory
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] := SRC1[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] := SRC1[127:0]
    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[MAXVL-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 *merging-masking* ; merging-masking
      IF *merging-masking*
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI;
ENDFOR

DEST[MAXVL-1:256] := 0

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

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

FOR j := 0 TO 3
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
VEXTRACTI128 (memory destination form)
CASE (imm8[0]) OF
  0: DEST[127:0] := SRC1[127:0]
ESAC.

VEXTRACTI128 (register destination form)
CASE (imm8[0]) OF
  0: DEST[127:0] := SRC1[127:0]
ESAC.

DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VEXTRACTI32x4 __m128i _mm512_extracti32x4_epi32(__m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_mask_extracti32x4_epi32(__m512i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_extracti32x8_epi32(__m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_mask_extracti32x8_epi32(__m512i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_maskz_extracti32x8_epi32(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_extracti64x2_epi64(__m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_mask_extracti64x2_epi64(__m512i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_maskz_extracti64x2_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_extracti64x4_epi64(__m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_mask_extracti64x4_epi64(__m512i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_maskz_extracti64x4_epi64(__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 Table 2-23, “Type 6 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-54, "Type E6NF Class Exception Conditions".
Additionally:
#UD IF VEX.L = 0.
#UD IF VEX.vvvv != 1111B or EVEX.vvvv != 1111B.
VFIXUPIMMPD—Fix Up Special Packed Float64 Values

### Instruction 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.128.66.0F3A.W1 S4 / r</td>
<td>A</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.256.66.0F3A.W1 S4 / r</td>
<td>A</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.512.66.0F3A.W1 S4 / r</td>
<td>A</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>Tuple Type</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>Full</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:rr/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[MAXVL-1:VL] := 0
Immediate Control Description:

![Immediate Control Description Diagram](image)

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

```
VFIXUPIMMPD __m512d _mm512_fixupimm_pd(__m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_pd(__mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_fixupimm_round_pd(__m512d a, __m512i tbl, int sae);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_round_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_round_pd(__mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m256d _mm256_fixupimm_pd(__m256d a, __m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_mask_fixupimm_pd(__m256d s, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_maskz_fixupimm_pd(__m256i c, int imm8);
VFIXUPIMMPD __m128d _mm_fixupimm_pd(__m128d a, __m128d b, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm_mask_fixupimm_pd(__m128d s, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm_maskz_fixupimm_pd(__m128i c, int imm8);
```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions”.
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.128.66.0F3A.W0 54 /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 54 /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 54 / r</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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

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[31:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j := 0;
        SNAN_TOKEN: j := 1;
        ZERO_VALUE_TOKEN: j := 2;
        POS_ONE_VALUE_TOKEN: j := 3;
        NEG_INF_TOKEN: j := 4;
        POS_INF_TOKEN: j := 5;
        NEG_VALUE_TOKEN: j := 6;
        POS_VALUE_TOKEN: j := 7;
    }
    ; end source special CASE(tsrc...)
    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;
        0111: dest[31:0] := -0;
        1000: dest[31:0] := +0;
        1001: dest[31:0] := -1;
        1010: dest[31:0] := +1;
        1011: dest[31:0] := ½;
        1100: dest[31:0] := 90.0;
        1101: dest[31:0] := PI/2;
        1110: dest[31:0] := MAX_FLOAT;
        1111: dest[31:0] := -MAX_FLOAT;
    }
    ; end of token_response CASE
; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
; end fault reporting
return dest[31:0];
)
; end of FIXUPIMM_SP()

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

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

![Diagram of Immediate Control Description]

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

VFXUPIMMPS __m512 _mm512_fixupimm_ps(__m512 a, __m512i tbl, int imm);
VFXUPIMMPS __m512 _mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFXUPIMMPS __m512 _mm512_maskz_fixupimm_ps(__mmask16 k, __m512 a, __m512i tbl, int imm);
VFXUPIMMPS __m512 _mm512_fixupimm_round_ps(__m512 a, __m512i tbl, int sae);
VFXUPIMMPS __m512 _mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFXUPIMMPS __m512 _mm512_maskz_fixupimm_round_ps(__mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFXUPIMMPS __m256 _mm256_fixupimm_ps(__m256 a, __m256 b, __m256i c, int imm8);
VFXUPIMMPS __m256 _mm256_mask_fixupimm_ps(__m256 a, __mmask8 k, __m256 b, __m256i c, int imm8);
VFXUPIMMPS __m256 _mm256_maskz_fixupimm_ps(__mmask8 k, __m256 a, __m256 b, __m256i c, int imm8);
VFXUPIMMPS __m128 _mm128_fixupimm_ps(__m128 a, __m128 b, __m128i c, int imm8);
VFXUPIMMPS __m128 _mm128_mask_fixupimm_ps(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8);
VFXUPIMMPS __m128 _mm128_maskz_fixupimm_ps(__mmask8 k, __m128 a, __m128 b, __m128i c, int imm8);

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions”.

---

5-118 Vol. 2C

VFXUPIMMPS—Fix Up Special Packed Float32 Values
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.LLIG.66.0F3A.W1 55/ ib</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/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, 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.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
}
```
```c
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()
```
VFIXUPIMMSD (EVEX encoded version)
IF k1[0] OR *no writemask*
THEN DEST[63:0] := FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[63:0] remains unchanged* 
ELSE  DEST[63:0] := 0 ; zeroing-masking
FI
Ft;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-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 Table 2-47, “Type E3 Class Exception Conditions”.

VFIXUPIMMSD—Fix Up Special Scalar Float64 Value
### VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

#### Instruction 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.LLIG.66.0F3A.W0 55 /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>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</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

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, 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.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] := ((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;
  0111: dest[31:0] := -0;
  1000: dest[31:0] := +0;
  1001: dest[31:0] := -1;
  1010: dest[31:0] := +1;
  1011: dest[31:0] := ½;
  1100: dest[31:0] := 90.0;
  1101: dest[31:0] := PI/2;
  1110: dest[31:0] := MAX_FLOAT;
  1111: dest[31:0] := -MAX_FLOAT;
} ; end of token_response CASE

; The required fault reporting from imm8 is extracted
; TOKENs are mutually exclusive and TOKENs priority defines the order.
; Multiple faults related to a single token can occur simultaneously.
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
} ; end fault reporting
return dest[31:0];
} ; end of FIXUPIMM_SP()
VFIXUPIMMSS (EVEX encoded version)

IF \text{k1[0]} OR *no writemask*
  THEN \text{DEST}[31:0] := \text{FIXUPIMM}_\text{SP}(\text{DEST}[31:0], \text{SRC1}[31:0], \text{SRC2}[31:0], \text{imm8}[7:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN \text{*DEST}[31:0] remains unchanged*
      ELSE \text{DEST}[31:0] := 0 ; zeroing-masking
    FI

\text{Fl}; \text{DEST}[127:32] := \text{SRC1}[127:32]
\text{DEST}[\text{MAXVL}-1:128] := 0

Immediate Control Description:

**Figure 5-12. VFIXUPIMMSS Immediate Control Description**

`+ \text{INF} \rightarrow \#IE`
`- \text{VE} \rightarrow \#IE`
`- \text{INF} \rightarrow \#IE`
`\text{SNaN} \rightarrow \#IE`
`\text{ONE} \rightarrow \#IE`
`\text{ONE} \rightarrow \#ZE`
`\text{ZERO} \rightarrow \#IE`
`\text{ZERO} \rightarrow \#ZE`

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

VFIXUPIMMSS __m128 _mm_fixupimm_ss(__m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_ss(__mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 _mm_fixupimm_round_ss(__m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_round_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_round_ss(__mmask8 k, __m128 a, __m128i tbl, int imm, int sae);

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Table 2-47, "Type E3 Class Exception Conditions".
## 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.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.W1 A8 /r VFMADD213PD xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.W1 B8 /r VFMADD231PD xmm1, xmm2, xmm3/m128</td>
<td>A</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.256.66.0F38.W1 98 /r VFMADD132PD ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.W1 A8 /r VFMADD213PD ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256</td>
<td>A</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.128.66.0F38.W1 98 /r VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</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.128.66.0F38.W1 A8 /r VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</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.128.66.0F38.W1 B8 /r VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</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.256.66.0F38.W1 98 /r VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</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.256.66.0F38.W1 A8 /r VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</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.256.66.0F38.W1 B8 /r VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</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.512.66.0F38.W1 98 /r VFMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B</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.512.66.0F38.W1 A8 /r VFMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B</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.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B</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>
## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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 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 an XMM register and encoded in reg_field. The second source operand is an XMM register and encoded in VEX.vvvv. The third source operand is an 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
\[
\text{MAXNUM} := 2
\]
Elseif (VEX.256)
\[
\text{MAXNUM} := 4
\]
Fi

For \(i = 0\) to MAXNUM-1 {
\[
n := 64*i;
\]
\[
\text{DEST}[n+63:n] := \text{RoundFPControl\_MXCSR(DEST}[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
\]
}

If (VEX.128) then
\[
\text{DEST}[\text{MAXVL-1:128}] := 0
\]
Elseif (VEX.256)
\[
\text{DEST}[\text{MAXVL-1:256}] := 0
\]
Fi

**VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)**

If (VEX.128) then
\[
\text{MAXNUM} := 2
\]
Elseif (VEX.256)
\[
\text{MAXNUM} := 4
\]
Fi

For \(i = 0\) to MAXNUM-1 {
\[
n := 64*i;
\]
\[
\text{DEST}[n+63:n] := \text{RoundFPControl\_MXCSR(SRC2}[n+63:n]*\text{DEST}[n+63:n] + SRC3[n+63:n])
\]
}

If (VEX.128) then
\[
\text{DEST}[\text{MAXVL-1:128}] := 0
\]
Elseif (VEX.256)
\[
\text{DEST}[\text{MAXVL-1:256}] := 0
\]
Fi

**VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)**

If (VEX.128) then
\[
\text{MAXNUM} := 2
\]
Elseif (VEX.256)
\[
\text{MAXNUM} := 4
\]
Fi

For \(i = 0\) to MAXNUM-1 {
\[
n := 64*i;
\]
\[
\text{DEST}[n+63:n] := \text{RoundFPControl\_MXCSR(SRC2}[n+63:n]*SRC3[n+63:n] + \text{DEST}[n+63:n])
\]
}

If (VEX.128) then
\[
\text{DEST}[\text{MAXVL-1:128}] := 0
\]
Elseif (VEX.256)
\[
\text{DEST}[\text{MAXVL-1:256}] := 0
\]
Fi
VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] :=
      RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
      THEN
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
      ELSE
        DEST[i+63:i] :=
        RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI
  ENDFOR
DEST[MAXVL-1:VL] := 0
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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] :=
        RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

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
    ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

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

THEN

\[ \text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);} \]
ELSE

\[ \text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);} \]

FI;

FOR \(j := 0\) TO \(KL-1\)

\[ i := j * 64 \]

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

THEN \( \text{DEST}[i+63:i] := \)

\[ \text{RoundFPControl}(\text{SRC2}[i+63:i]\times\text{SRC3}[i+63:i] + \text{DEST}[i+63:i]) \]

ELSE

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

THEN \(*\text{DEST}[i+63:i]\) remains unchanged*\]
ELSE \(*\text{zeroing-masking} \)

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

FI

ENDFOR

\[ \text{DEST}[\text{MAXVL-1:VL}] := 0 \]

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR \(j := 0\) TO \(KL-1\)

\[ i := j * 64 \]

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

THEN

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

\[ \text{THEN} \]

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

\[ \text{RoundFPControl_MXCSR(\text{SRC2}[i+63:i]\times\text{SRC3}[63:0] + \text{DEST}[i+63:i])} \]

ELSE

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

\[ \text{RoundFPControl_MXCSR(\text{SRC2}[i+63:i]\times\text{SRC3}[i+63:i] + \text{DEST}[i+63:i])} \]

FI;
ELSE

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

THEN \(*\text{DEST}[i+63:i]\) remains unchanged*\]
ELSE \(*\text{zeroing-masking} \)

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

FI

ENDFOR

\[ \text{DEST}[\text{MAXVL-1:VL}] := 0 \]
Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d _mm128d_mm_mask_fmaddd_pd(__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d _mm256d_mm_mask_fmaddd_pd(__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.
### 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.128.66.0F38.W0 98 /r VFMADD132PS xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.W0 A8 /r VFMADD213PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 B8 /r VFMADD231PS xmm1, xmm2, xmm3/m128</td>
<td>A</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.256.66.0F38.W0 98 /r VFMADD132PS ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.W0 A8 /r VFMADD213PS ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.W0 B8 /r VFMADD231PS ymm1, ymm2, ymm3/m256</td>
<td>A</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.128.66.0F38.W0 98 /r VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 A8 /r VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m32bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 B8 /r VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 98 /r VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A8 /r VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m32bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 B8 /r VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 98 /r VFMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A8 /r VFMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m32bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 B8 /r VFMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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 first source operand is also the destination 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 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 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 four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions**: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version**: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version**: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADD132PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM := 4
ELSEIF (VEX.256)
   MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
   n := 32*i;
   DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI

**VFMADD213PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM := 4
ELSEIF (VEX.256)
   MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
   n := 32*i;
   DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI

**VFMADD231PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM := 4
ELSEIF (VEX.256)
   MAXNUM := 8
FI
For i = 0 to MAXNUM-1 {
   n := 32*i;
   DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI
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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
    FOR j := 0 TO KL-1
        i := j * 32
        IF k1[j] OR *no writemask*
            THEN DEST[i+31:i] :=
                RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+31:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] := 0

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
    FOR j := 0 TO KL-1
        i := j * 32
        IF k1[j] OR *no writemask*
            THEN
                IF (EVEX.b = 1)
                    THEN
                        Dest[i+31:i] :=
                            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                    ELSE
                        Dest[i+31:i] :=
                            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE ; zeroing-masking
                        Dest[i+31:i] := 0
                FI
        FI;
    ENDFOR
    Dest[MAXVL-1:VL] := 0
VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[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
        ENDIF
    ENDIF
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
        ELSE
            SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
        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;
    FOR j := 0 TO KL-1
        i := j * 32
        IF k1[j] OR *no writemask*
            THEN
                IF (EVEX.b = 1)
                    THEN
                        DEST[i+31:i] :=
                            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                    ELSE
                        DEST[i+31:i] :=
                            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                    FI;
                ELSE
                    IF *merging-masking* ; merging-masking
                        THEN *DEST[i+31:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+31:i] := 0
                    FI
                FI;
    ENDFOR
    DEST[MAXVL-1:VL] := 0

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] := 0
                FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFMADDxxxPS __m512 __mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_fmask_fadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_maskz_fadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_mask3_fadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 __mm512_mask_fadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_maskz_fadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask3_fadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 __mm256_mask_fadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_maskz_fadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_mask3_fadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 __mm128_mask_fadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm128_maskz_fadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm128_mask3_fadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 __mm128_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 __mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.
VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar 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>VEX.LIG.66.0F38.W1 99 /r \n VFMADD132SD xmm1, xmm2, xmm3/m64</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 A9 /r \n VFMADD213SD xmm1, xmm2, xmm3/m64</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 B9 /r \n VFMADD231SD xmm1, xmm2, xmm3/m64</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 99 /r \n VFMADD132SD xmm1 (k1)(z), xmm2, xmm3/m64[er]</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 A9 /r \n VFMADD213SD xmm1 (k1)(z), xmm2, xmm3/m64[er]</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 B9 /r \n VFMADD231SD xmm1 (k1)(z), xmm2, xmm3/m64[er]</td>
<td>B 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>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
```

ELSE

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
```

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[MAXVL-1:128] := 0
```

**VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)**

If (EVEX.b = 1) and SRC3 *is a register*

THEN

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
```

ELSE

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
```

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[MAXVL-1:128] := 0
```
VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF k1[0] or *no writemask*
    THEN 
        DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
        FI;
    FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := MAXVL-1:128RoundFPControl_MXCSR(Dest[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[MAXVL-1:128] := 0

VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + SRC3[63:0])
DEST[MAXVL-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[MAXVL-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_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d _mm_mask_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_mask_flag_s(__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”.
**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 Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W0 99 / r</td>
<td>A</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>VFMADD132SS xmm1, xmm2, xmm3/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W0 A9 / r</td>
<td>A</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>VFMADD213SS xmm1, xmm2, xmm3/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W0 B9 / r</td>
<td>A</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>VFMADD231SS xmm1, xmm2, xmm3/m32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.66.0F38.W0 99 / r</td>
<td>B</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>VFMADD132SS xmm1{K1}{Z}, xmm2, xmm3/m32{ER}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.66.0F38.W0 A9 / r</td>
<td>B</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>VFMADD213SS xmm1{K1}{Z}, xmm2, xmm3/m32{ER}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.66.0F38.W0 B9 / r</td>
<td>B</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>
<tr>
<td>VFMADD231SS xmm1{K1}{Z}, xmm2, xmm3/m32{ER}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:rm (r)</td>
<td>NA</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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    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[MAXVL-1:128] := 0

**VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    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[MAXVL-1:128] := 0
**VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
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[MAXVL-1:128] := 0

**VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)**

DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[MAXVL-1:128] := 0

**VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)**

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[MAXVL-1:128] := 0

**VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)**

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[MAXVL-1:128] := 0

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

VFMADDxxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Inexact, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”.

---

5-144 Vol. 2C  VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single-Precision Floating-Point Values
### 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 Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W1 96 / r VFMADDSUB132PD xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.W1 A6 / r VFMADDSUB213PD xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.W1 B6 / r VFMADDSUB231PD xmm1, xmm2, xmm3/m128</td>
<td>A</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.256.66.0F38.W1 96 / r VFMADDSUB213PDymm1, xmm2, ymm3/m256</td>
<td>A</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.256.66.0F38.W1 A6 / r VFMADDSUB231PDymm1, xmm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 B6 / r VFMADDSUB231PDymm1, xmm2, ymm3/m256</td>
<td>A</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.128.66.0F38.W1 96 / r EVEX.128.66.0F38.W1 A6 / r EVEX.128.66.0F38.W1 B6 / r VFMADDSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</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.256.66.0F38.W1 96 / r EVEX.256.66.0F38.W1 A6 / r EVEX.256.66.0F38.W1 B6 / r VFMADDSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</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.256.66.0F38.W1 96 / r EVEX.256.66.0F38.W1 A6 / r EVEX.256.66.0F38.W1 B6 / r VFMADDSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</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>
</tbody>
</table>
VFMADDSUB132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.
VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANS are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMADDSUB132PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)

DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
FI

**VFMADDSUB213PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)

DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
FI

**VFMADDSUB231PD DEST, SRC2, SRC3**

IF (VEX.128) THEN

DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)

DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] := RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
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

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

ELSE

\[ \text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC)}; \]

FI;

FOR \( j := 0 \) TO \( KL-1 \)

\[ i := j \times 64 \]

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

THEN

IF \( j \) *is even*

THEN \( \text{DEST}[i+63:i] := \)\[ \text{RoundFPControl(DEST}[i+63:i]\times SRC3[i+63:i] - SRC2[i+63:i])} \]

ELSE \( \text{DEST}[i+63:i] := \)\[ \text{RoundFPControl(DEST}[i+63:i]\times SRC3[i+63:i] + SRC2[i+63:i])} \]

FI

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

\( \text{DEST}[i+63:i] := 0 \)

FI

ENDFOR

\( \text{DEST}[\text{MAXVL-1:VL}] := 0 \)
VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] :=
                RoundFPControl_MXCSR(DEST[i+63:i] * SRC3[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 (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
  ELSE
    ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI
    ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
            ELSE
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
            ELSE
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
          FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                ELSE DEST[i+63:i] :=
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*;
                merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE
                zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] :=
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
            ELSE
                DEST[i+63:i] :=
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - Dest[i+63:i])
        FI;
      ELSE
        IF (EVEX.b = 1)
          THEN
            DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
          ELSE
            DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
        FI;
    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[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUBxxxPD __m512d __m512d __m512d __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __m512d __m512d __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __m512d __m512d __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __m512d __m512d __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __m512d __m512d __m512d a, __m512d b, __m512d c, int r);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.

Intrinsic equivalent:

VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alernating Add/Subtract of Packed Double-Precision
## 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.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 B6 /r VFMADDSUB213PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 96 /r VFMADDSUB231PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 B6 /r VFMADDSUB213PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 96 /r VFMADDSUB213PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 B6 /r VFMADDSUB231PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 96 /r VFMADDSUB231PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst{er}</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>
VFMADDSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.
Operation
In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMADDUB132PS DEST, SRC2, SRC3**
- IF (VEX.128) THEN
  - MAXNUM := 2
- ELSEIF (VEX.256)
  - MAXNUM := 4
- FI
- For i = 0 to MAXNUM -1{
  - \( n := 64i \);
  - \( \text{DEST}[n+31:n] := \text{RoundFPControl\_MXCSR(DEST}[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n]) \)
  - \( \text{DEST}[n+63:n+32] := \text{RoundFPControl\_MXCSR(DEST}[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32]) \)
- } IF (VEX.128) THEN
  - \( \text{DEST}[\text{MAXVL}-1:128] := 0 \)
- ELSEIF (VEX.256)
  - \( \text{DEST}[\text{MAXVL}-1:256] := 0 \)
- FI

**VFMADDUB213PS DEST, SRC2, SRC3**
- IF (VEX.128) THEN
  - MAXNUM := 2
- ELSEIF (VEX.256)
  - MAXNUM := 4
- FI
- For i = 0 to MAXNUM -1{
  - \( n := 64i \);
  - \( \text{DEST}[n+31:n] := \text{RoundFPControl\_MXCSR(SRC2}[n+31:n]*\text{DEST}[n+31:n] - SRC3[n+31:n]) \)
  - \( \text{DEST}[n+63:n+32] := \text{RoundFPControl\_MXCSR(SRC2}[n+63:n+32]*\text{DEST}[n+63:n+32] + SRC3[n+63:n+32]) \)
- } IF (VEX.128) THEN
  - \( \text{DEST}[\text{MAXVL}-1:128] := 0 \)
- ELSEIF (VEX.256)
  - \( \text{DEST}[\text{MAXVL}-1:256] := 0 \)
- FI
VFMADDSUB231PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI
    ENDFOR
    DEST[MAXVL-1:VL] := 0
VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] :=
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
            ELSE
                DEST[i+31:i] :=
                RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                  DEST[i+31:i] :=
                  RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
              ELSE
                  DEST[i+31:i] :=
                  RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
              FI;
          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
  ENDFOR

DEST[MAXVL-1:VL] := 0
VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF j *is even*
    THEN
      DEST[i+31:i] :=
      RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
    ELSE
      DEST[i+31:i] :=
      RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
    FI
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (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[31:0])
    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;

5-160  Vol. 2C  VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] := 0
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    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[MAXVL-1:VL] := 0
VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] :=
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
              ELSE
              DEST[i+31:i] :=
              RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[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
        DEST[i+31:i] := 0
    FI
  ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUBxxxPS __m512 __m512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __m512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __m512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __m512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 __m512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512 __m512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __m512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 __m512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxxPS __m256 __m256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __m256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 __m256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128 __m128_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __m128_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 __m128_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m256 __m256_fmaddsub_ps(__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.
### 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.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1, xmm2, xmm3/m128</td>
<td>A 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.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1, xmm2, xmm3/m128</td>
<td>A 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.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1, xmm2, xmm3/m128</td>
<td>A 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.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1, ymm2, ymm3/m256</td>
<td>A 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.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1, ymm2, ymm3/m256</td>
<td>A 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.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1, ymm2, ymm3/m256</td>
<td>A 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.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B 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.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B 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.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B 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.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B 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.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B 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).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.
VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMSUBADD132PD DEST, SRC2, SRC3**

If (VEX.128) THEN

\[
\text{DEST}[63:0] := \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[63:0]*\text{SRC3}[63:0] + \text{SRC2}[63:0])
\]

\[
\text{DEST}[127:64] := \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[127:64]*\text{SRC3}[127:64] - \text{SRC2}[127:64])
\]

\[
\text{DEST}[:128] := 0
\]

ELSEIF (VEX.256)

\[
\text{DEST}[63:0] := \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[63:0]*\text{SRC3}[63:0] + \text{SRC2}[63:0])
\]

\[
\text{DEST}[127:64] := \text{RoundFPControl}_\text{MXCSR}(\text{DEST}[127:64]*\text{SRC3}[127:64] - \text{SRC2}[127:64])
\]

\[
\text{DEST}[:128] := 0
\]

FI

**VFMSUBADD213PD DEST, SRC2, SRC3**

If (VEX.128) THEN

\[
\text{DEST}[63:0] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[63:0]*\text{DEST}[63:0] + \text{SRC3}[63:0])
\]

\[
\text{DEST}[127:64] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[127:64]*\text{SRC3}[127:64] - \text{DEST}[127:64])
\]

\[
\text{DEST}[:128] := 0
\]

ELSEIF (VEX.256)

\[
\text{DEST}[63:0] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[63:0]*\text{SRC3}[63:0] + \text{DEST}[63:0])
\]

\[
\text{DEST}[127:64] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[127:64]*\text{SRC3}[127:64] - \text{DEST}[127:64])
\]

\[
\text{DEST}[:128] := 0
\]

FI

**VFMSUBADD231PD DEST, SRC2, SRC3**

If (VEX.128) THEN

\[
\text{DEST}[63:0] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[63:0]*\text{SRC3}[63:0] + \text{DEST}[63:0])
\]

\[
\text{DEST}[127:64] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[127:64]*\text{SRC3}[127:64] - \text{DEST}[127:64])
\]

\[
\text{DEST}[:128] := 0
\]

ELSEIF (VEX.256)

\[
\text{DEST}[63:0] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[63:0]*\text{SRC3}[63:0] + \text{DEST}[63:0])
\]

\[
\text{DEST}[127:64] := \text{RoundFPControl}_\text{MXCSR}(\text{SRC2}[127:64]*\text{SRC3}[127:64] - \text{DEST}[127:64])
\]

\[
\text{DEST}[:128] := 0
\]

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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);

ELSE

    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);

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

ENDFOR

DEST[MAXVL-1:VL] := 0
VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] :=
                                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] :=
                                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[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;
                ELSE
                    IF *merging-masking* ; merging-masking
                        THEN *DEST[i+63:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+63:i] := 0
                    FI
            FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN DEST[i+63:i] :=
          RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
        ELSE DEST[i+63:i] :=
          RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
      FI
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] :=
              RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
            ELSE
                DEST[i+63:i] :=
                RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
        FI;
      ELSE
        IF (EVEX.b = 1)
          THEN
            DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
          ELSE
              DEST[i+63:i] :=
              RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  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
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] :=
              RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
            ELSE
              DEST[i+63:i] :=
              RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] :=
              RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
            ELSE
              DEST[i+63:i] :=
              RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
        FI
      FI
ENDFOR
DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUBADDxxxPD __m512d __mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m256d __mm256_fmsubadd_pd(__m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_mask_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m128d __mm128_fmsubadd_pd(__m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm128_mask_fmsubadd_pd(__mmask8 k, __m128d a, __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_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);
VFMSUBADDxxxPD __m128d __mm128_mask3_fmsubadd_round_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.
## 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.128.66.F38.W0 97 /r VFMSUBADD132PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.F38.W0 A7 /r VFMSUBADD213PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>VEX.128.66.F38.W0 B7 /r VFMSUBADD231PS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>VEX.256.66.F38.W0 97 /r VFMSUBADD132PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.F38.W0 A7 /r VFMSUBADD213PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>VEX.256.66.F38.W0 B7 /r VFMSUBADD231PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.66.F38.W0 97 /r VFMSUBADD132PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.66.F38.W0 A7 /r VFMSUBADD213PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.66.F38.W0 B7 /r VFMSUBADD231PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.F38.W0 97 /r VFMSUBADD132PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.F38.W0 A7 /r VFMSUBADD213PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.F38.W0 B7 /r VFMSUBADD231PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.512.66.F38.W0 97 /r VFMSUBADD132PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.512.66.F38.W0 A7 /r VFMSUBADD213PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.512.66.F38.W0 B7 /r VFMSUBADD231PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>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>
</tr>
</tbody>
</table>
Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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
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).

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 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANS are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.
**Operation**

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFMSUBADD132PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM := 2
ELSEIF (VEX.256)
   MAXNUM := 4
FI

For i = 0 to MAXNUM -1{
   n := 64*i;
   DEST[n+31:n] := RoundFPControl_MXCSR(DST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}

IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI

**VFMSUBADD213PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM := 2
ELSEIF (VEX.256)
   MAXNUM := 4
FI

For i = 0 to MAXNUM -1{
   n := 64*i;
   DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}

IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI

**VFMSUBADD231PS DEST, SRC2, SRC3**

IF (VEX.128) THEN
   MAXNUM := 2
ELSEIF (VEX.256)
   MAXNUM := 4
FI

For i = 0 to MAXNUM -1{
   n := 64*i;
   DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}

IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI
VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
        IF j *is even*
            THEN DEST[i+31:i] :=
                RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            ELSE DEST[i+31:i] :=
                RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI
        ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0
VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
            ELSE
                DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
                DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
            ELSE
                DEST[i+31:i] :=
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI;
        ELSE
          IF *merging-masking*
            THEN *
                DEST[i+31:i] remains unchanged*
            ELSE
                DEST[i+31:i] := 0
          FI
      FI
  ELSE
    FI
ENDFOR

DEST[MAXVL-1:VL] := 0
VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                ELSE DEST[i+31:i] :=
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI
    ENDFOR

DEST[MAXVL-1:VL] := 0
VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
                DEST[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
  ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    DEST[i+31:i] :=
                        RoundFPControl(SRC2[i+31:i] * SRC3[i+31:i] + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] :=
                        RoundFPControl(SRC2[i+31:i] * SRC3[i+31:i] - DEST[i+31:i])
                FI
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+31:i] := 0
                FI
        FI
ENDFOR

DEST[MAXVL-1:VL] := 0
VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
    IF j *is even*
      THEN
        IF (EVEX.b = 1)
          THEN
            DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
          ELSE
            DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
        FI;
      ELSE
        IF (EVEX.b = 1)
          THEN
            DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
          ELSE
            DEST[i+31:i] :=
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
        FI;
      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
  ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADDxxxPS __m512 __mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 __mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512 __mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 __mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m256 __mm256_mask_fmsubadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 __mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 __mm256_mask3_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBADDxxxPS __m128 __mm128_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 __mm128_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 __mm128_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBADDxxxPS __m128 __mm128_fmsubadd_ps(__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 __mm256_fmsubadd_ps(__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.
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 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFMSUB132PD xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VFMSUB213PD xmm1, xmm2, xmm3/m128</td>
<td>A 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>VFMSUB231PD xmm1, xmm2, xmm3/m128</td>
<td>A 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>VFMSUB132PD ymm1, ymm2, ymm3/m256</td>
<td>A 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>VFMSUB213PD ymm1, ymm2, ymm3/m256</td>
<td>A 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>VFMSUB231PD ymm1, ymm2, ymm3/m256</td>
<td>A 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>VEX.128.66.0F38.0F W1 9A /r</td>
<td>B 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>VEX.128.66.0F38.0F W1 AA /r</td>
<td>B 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>VEX.128.66.0F38.0F W1 BA /r</td>
<td>B 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>VEX.256.66.0F38.0F W1 9A /r</td>
<td>B 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>VEX.256.66.0F38.0F W1 AA /r</td>
<td>B 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>VEX.256.66.0F38.0F W1 BA /r</td>
<td>B 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>VEX.512.66.0F38.0F W1 9A /r</td>
<td>B V/V</td>
<td>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>VEX.512.66.0F38.0F W1 AA /r</td>
<td>B V/V</td>
<td>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>VEX.512.66.0F38.0F W1 BA /r</td>
<td>B V/V</td>
<td>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>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMSUB132PD**: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB213PD**: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUB231PD**: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions**: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version**: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

**VEX.128 encoded version**: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
Operation
In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PD DEST, SRC2, SRC3 (VEX encoded versions)
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMSUB213PD DEST, SRC2, SRC3 (VEX encoded versions)
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

VFMSUB231PD DEST, SRC2, SRC3 (VEX encoded versions)
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    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
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    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
                        FI
                    ELSE
                        IF *merging-masking* ; merging-masking
                            THEN *DEST[i+63:i] remains unchanged*
                        ELSE ; zeroing-masking
                            DEST[i+63:i] := 0
                        FI
                FI
            ELSE
                FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)

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

\[
\text{IF } (VL = 512) \text{ AND } (EVEX.b = 1) \text{ THEN}
\]

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

\[
\text{ELSE}
\]

\[
\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC)};
\]

\[
\text{FI};
\]

\[
\text{FOR } j := 0 \text{ TO } KL-1
\]

\[
i := j \times 64
\]

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

\[
\text{THEN } \text{DEST}[i+63:i] := \text{RoundFPControl}(\text{SRC2}[i+63:i] \times \text{DEST}[i+63:i] - \text{SRC3}[i+63:i])
\]

\[
\text{ELSE}
\]

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

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

\[
\text{ELSE} \text{; zeroing-masking}
\]

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

\[
\text{FI}
\]

\[
\text{FI};
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAXVL-1:VL}] := 0
\]

VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

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

\[
\text{FOR } j := 0 \text{ TO } KL-1
\]

\[
i := j \times 64
\]

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

\[
\text{THEN}
\]

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

\[
\text{THEN}
\]

\[
\text{DEST}[i+63:i] := \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[i+63:i] \times \text{DEST}[i+63:i] - \text{SRC3}[63:0])
\]

\[
+31:i)
\]

\[
\text{ELSE}
\]

\[
\text{DEST}[i+63:i] := \text{RoundFPControl}_{\text{MXCSR}}(\text{SRC2}[i+63:i] \times \text{DEST}[i+63:i] - \text{SRC3}[i+63:i])
\]

\[
\text{FI};
\]

\[
\text{ELSE}
\]

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

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

\[
\text{ELSE} \text{; zeroing-masking}
\]

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

\[
\text{FI}
\]

\[
\text{FI};
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST}[\text{MAXVL-1:VL}] := 0
\]
VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] :=
        RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
            ELSE
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxPD __m512d _mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d _mm256_fmsub_pd(__m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_fmsub_round_pd(__m256d a, __m256d b, __m256d c, int r);
VFMSUBxxxPD __m256d _mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m256d _mm256_fmsub_round_pd(__m256d a, __m256d b, __m256d c, int r);
VFMSUBxxxPD __m128d _mm_fmsub_pd(__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m256d _mm256_fmsub_pd(__m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m128d _mm_mask_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_fmsub_round_pd(__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m256d _mm256_fmsub_round_pd(__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/E n</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 9A /r VFMSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>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>
</tr>
<tr>
<td>VEX.128.66.0F38.w0 AA /r VFMSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>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>
</tr>
<tr>
<td>VEX.128.66.0F38.w0 BA /r VFMSUB231PS xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>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>
</tr>
<tr>
<td>VEX.256.66.0F38.w0 9A /r VFMSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>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>
</tr>
<tr>
<td>VEX.256.66.0F38.w0 AA /r VFMSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>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>
</tr>
<tr>
<td>VEX.256.66.0F38.0 BA /r VFMSUB231PS ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w0 9A /r VFMSUB132PS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.66.0F38.w0 AA /r VFMSUB213PS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.66.0F38.w0 BA /r VFMSUB231PS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w0 9A /r VFMSUB132PS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.0F38.w0 AA /r VFMSUB213PS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.0F38.w0 BA /r VFMSUB231PS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 9A /r VFMSUB132PS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst[er]</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 AA /r VFMSUB213PS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst[er]</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 BA /r VFMSUB231PS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst[er]</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>
VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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-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 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 first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.
**Operation**

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN  
    MAXNUM := 2  
ELSEIF (VEX.256)  
    MAXNUM := 4  
FI  
For i = 0 to MAXNUM-1 {  
    n := 32*i;  
    DEST[n+31:n] := RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])  
}  
IF (VEX.128) THEN  
    DEST[MAXVL-1:128] := 0  
ELSEIF (VEX.256)  
    DEST[MAXVL-1:256] := 0  
FI

**VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN  
    MAXNUM := 2  
ELSEIF (VEX.256)  
    MAXNUM := 4  
FI  
For i = 0 to MAXNUM-1 {  
    n := 32*i;  
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])  
}  
IF (VEX.128) THEN  
    DEST[MAXVL-1:128] := 0  
ELSEIF (VEX.256)  
    DEST[MAXVL-1:256] := 0  
FI

**VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN  
    MAXNUM := 2  
ELSEIF (VEX.256)  
    MAXNUM := 4  
FI  
For i = 0 to MAXNUM-1 {  
    n := 32*i;  
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])  
}  
IF (VEX.128) THEN  
    DEST[MAXVL-1:128] := 0  
ELSEIF (VEX.256)  
    DEST[MAXVL-1:256] := 0  
FI
VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] :=
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[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[MAXVL-1:VL] := 0
VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

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

\[\begin{array}{l}
\text{THEN} \\
\quad \text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);} \\
\text{ELSE} \\
\quad \text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);} \\
\end{array}\]

\[\begin{array}{l}
\text{Fi;} \\
\text{FOR } j := 0 \text{ TO } KL-1 \\
\quad i := j \times 32 \\
\quad \text{IF } k1[j] \text{ OR *no writemask*} \\
\text{THEN } \text{DEST}[i+31:i] := \\
\quad \text{RoundFPControl_MXCSR(SRC2}[i+31:i]*\text{DEST}[i+31:i] - \text{SRC3}[i+31:i]) \\
\text{ELSE} \\
\quad \text{IF *merging-masking*} \\
\text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged*} \\
\quad \text{ELSE} \\
\quad \text{DEST}[i+31:i] := 0 \\
\text{Fi} \\
\end{array}\]

\*[merging-masking]

\*[merge-masking]

\*[zeroing-masking]

\*[zero-masking]

\[\begin{array}{l}
\text{Fi} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAXVL-1:VL}] := 0 \\
\end{array}\]

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

\[\begin{array}{l}
\text{FOR } j := 0 \text{ TO } KL-1 \\
\quad i := j \times 32 \\
\quad \text{IF } k1[j] \text{ OR *no writemask*} \\
\text{THEN} \\
\quad \text{IF (EVEX.b = 1)} \\
\text{THEN} \\
\quad \text{DEST}[i+31:i] := \\
\quad \text{RoundFPControl_MXCSR(SRC2}[i+31:i]*\text{DEST}[i+31:i] - \text{SRC3}[31:0]) \\
\text{ELSE} \\
\quad \text{DEST}[i+31:i] := \\
\quad \text{RoundFPControl_MXCSR(SRC2}[i+31:i]*\text{DEST}[i+31:i] - \text{SRC3}[i+31:i]) \\
\text{Fi;} \\
\text{ELSE} \\
\quad \text{IF *merging-masking*} \\
\text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged*} \\
\text{ELSE} \\
\quad \text{DEST}[i+31:i] := 0 \\
\text{Fi} \\
\end{array}\]

\*[merging-masking]

\*[merge-masking]

\*[zeroing-masking]

\*[zero-masking]

\[\begin{array}{l}
\text{Fi} \\
\text{ENDFOR} \\
\text{DEST}[\text{MAXVL-1:VL}] := 0 \\
\end{array}\]
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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   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*
           THEN *DEST[i+31:i] remains unchanged*
       ELSE
           DEST[i+31:i] := 0
       FI
   FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j := 0 TO KL-1
i := j * 32
IF k1[j] OR *no writemask*
   THEN
       IF (EVEX.b = 1)
           THEN
               DEST[i+31:i] :=
                   RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
               ELSE
                   DEST[i+31:i] :=
                       RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                   FI;
       ELSE
           IF *merging-masking*
               THEN *DEST[i+31:i] remains unchanged*
           ELSE
               DEST[i+31:i] := 0
           FI
   FI;
ENDFOR

DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFMSUBxxxPS __m512 __mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m256 __mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm128_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm128_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm128_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm128_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm128_fmsub_round_ps(__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm128_mask_fmsub_round_ps(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxPS __m128 __mm128_maskz_fmsub_round_ps(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxPS __m128 __mm128_mask3_fmsub_round_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.

VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   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[MAXVL-1:128] := 0

**VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)**
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
       SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   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[MAXVL-1:128] := 0
VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
THEN
    DEST[63:0] := RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        THEN DEST[63:0] := 0
    FI;
FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(Dest[63:0]*SRC2[63:0] - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*Dest[63:0] - SRC3[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)
DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - Dest[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxSD _m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD _m128d _mm_mask_fmsub_sd(__m128d a, __m128d b, __m128d c);
VFMSUBxxxSD _m128d _mm_maskz_fmsub_sd(__m128d a, __m128d b, __m128d c);
VFMSUBxxxSD _m128d _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD _m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD _m128d _mm_maskz_fmsub_round_sd(__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
VEE-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”.

5-200 Vol. 2C VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values
VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</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-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

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

ELSE

\[
\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC)};
\]

FI;

IF k1[0] or *no writemask*

THEN

\[
\text{DEST}[31:0] := \text{RoundFPControl}(\text{DEST}[31:0] * \text{SRC3}[31:0] - \text{SRC2}[31:0])
\]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[31:0] := 0

FI;

FI;

\[
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]

**VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

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

ELSE

\[
\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC)};
\]

FI;

IF k1[0] or *no writemask*

THEN

\[
\text{DEST}[31:0] := \text{RoundFPControl}(\text{SRC2}[31:0] * \text{DEST}[31:0] - \text{SRC3}[31:0])
\]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[31:0] remains unchanged*

ELSE ; zeroing-masking

THEN DEST[31:0] := 0

FI;

FI;

\[
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
   THEN
     SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
     SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   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[MAXVL-1:128] := 0
VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[MAXVL-1:128] := 0
VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[MAXVL-1:128] := 0
VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(___m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(___m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_ss(___m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”.

VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values
### 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.128.66.0F38.W1 9C /r VFNMADD132PD xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.W1 AC /r VFNMADD213PD xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.W1 BC /r VFNMADD231PD xmm1, xmm2, xmm3/m128</td>
<td>A</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.256.66.0F38.W1 9C /r VFNMADD132PD ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.W1 AC /r VFNMADD213PD ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.W1 BC /r VFNMADD231PD ymm1, ymm2, ymm3/m256</td>
<td>A</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.128.66.0F38.W1 9C /rVFNMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst</td>
<td>B</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.128.66.0F38.W1 AC /rVFNMADD213PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst</td>
<td>B</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.128.66.0F38.W1 BC /rVFNMADD231PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst</td>
<td>B</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.256.66.0F38.W1 9C /rVFNMADD132PD ymm0 {k1}{z}, ymm1, ymm2/m256/m64bcst</td>
<td>B</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.256.66.0F38.W1 AC /rVFNMADD213PD ymm0 {k1}{z}, ymm1, ymm2/m256/m64bcst</td>
<td>B</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.256.66.0F38.W1 BC /rVFNMADD231PD ymm0 {k1}{z}, ymm1, ymm2/m256/m64bcst</td>
<td>B</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.512.66.0F38.W1 9C /rVFNMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B</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.512.66.0F38.W1 AC /rVFNMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B</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.512.66.0F38.W1 BC /rVFNMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B</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/VFNADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

**Instruction Set Reference, V-Z**

**Vol. 2C 5-205**

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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**

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[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI
VFNMADD213PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
   MAXNUM := 2
ELSEIF (VEX.256)
   MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
   n := 64*i;
   DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI
VFNMADD231PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
   MAXNUM := 2
ELSEIF (VEX.256)
   MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
   n := 64*i;
   DEST[n+63:n] := RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI
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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
  SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] :=
      RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

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[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[MAXVL-1:VL] := 0
VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] :=
        RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[63:0])
            ELSE
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

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[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_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 __m256d _mm256_mask fnmadd pd (__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.

5-210  Vol. 2C  VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values
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.128.66.0F38.w0 9C /r VFNMADD132PS xmm1, xmm2, xmm3/m128</td>
<td>A</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 add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.w0 AC /r VFNMADD213PS xmm1, xmm2, xmm3/m128</td>
<td>A</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 add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.w0 BC /r VFNMADD231PS xmm1, xmm2, xmm3/m128</td>
<td>A</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 add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.w0 9C /r VFNMADD132PS ymm1, ymm2, ymm3/m256</td>
<td>A</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 add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.w0 AC /r VFNMADD213PS ymm1, ymm2, ymm3/m256</td>
<td>A</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 add to ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.0 BC /r VFNMADD231PS ymm1, ymm2, ymm3/m256</td>
<td>A</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 and add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w0 9C /r VFNMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</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 add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w0 AC /r VFNMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</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 add to xmm3/m128/m32bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w0 BC /r VFNMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</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 and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w0 9C /r VFNMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</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 add to ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w0 AC /r VFNMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</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 add to ymm3/m256/m32bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w0 BC /r VFNMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</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 and add to ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 9C /r VFNMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst(er)</td>
<td>B</td>
<td>V/V</td>
<td>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>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 AC /r VFNMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst(er)</td>
<td>B</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 add to zmm3/m512/m32bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 BC /r VFNMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst(er)</td>
<td>B</td>
<td>V/V</td>
<td>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>
</tr>
</tbody>
</table>
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 the 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).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, “*” and “+” symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**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[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI
VFNMADD213PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI
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[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
      RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
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[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[MAXVL-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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
    RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
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  
ENDFOR  
DEST[MAXVL-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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);  
ELSE  
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
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  
ENDIF  
ENDFOR  
DEST[MAXVL-1:VL] := 0
VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1)
    THEN
      DEST[i+31:i] := RoundFPControl_MUXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
      ELSE
        DEST[i+31:i] := RoundFPControl_MUXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxPS __m512 __m512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 __m512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 __m512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512 __m512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 __m512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512 __m512_mask_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 __m512_maskz_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m256 __m256_mask_fnmadd_ps(__m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 __m256_maskz_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 __m256_mask3_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m128 __m128_mask_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 __m128_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 __m128_mask3_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m256 __m256_fnmadd_ps (__m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 __m256_fnmadd_round_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.

VFNMADD231PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

**Operands**

<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.LG.66.0F38.W1 9D /r VFNMADD132SD xmm1, xmm2, xmm3/m64</td>
<td>A V/V 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>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LG.66.0F38.W1 AD /r VFNMADD213SD xmm1, xmm2, xmm3/m64</td>
<td>A V/V 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>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.LG.66.0F38.W1 BD /r VFNMADD231SD xmm1, xmm2, xmm3/m64</td>
<td>A V/V 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>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLG.66.0F38.W1 9D /r VFNMADD132SD xmm1[k1]{k1}{z}, xmm2, xmm3/m64[er]</td>
<td>B V/V AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLG.66.0F38.W1 AD /r VFNMADD213SD xmm1[k1]{k1}{z}, xmm2, xmm3/m64[er]</td>
<td>B V/V AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLG.66.0F38.W1 BD /r VFNMADD231SD xmm1[k1]{k1}{z}, xmm2, xmm3/m64[er]</td>
<td>B V/V AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Op/En Tuple Type Operand 1 Operand 2 Operand 3 Operand 4**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</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**

VFNMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low quadword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

**Operation**

In the operations below, "*" and "*+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

**VFNMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
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[MAXVL-1:128] := 0

**VFNMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
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[MAXVL-1:128] := 0
VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[63:0] := 0
        FI;
    FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

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[MAXVL-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[MAXVL-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[MAXVL-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, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEX-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”.
VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar 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>VEX.LIG.66.0F38.W0 9D /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>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.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W0 AD /r</td>
<td>A</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 add to xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W0 BD /r</td>
<td>A</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 add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W0 9D /r</td>
<td>B</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 add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W0 AD /r</td>
<td>B</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 add to xmm3/m32 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W0 BD /r</td>
<td>B</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 add to xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</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

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 first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
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[MAXVL-1:128] := 0

VFNMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
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[MAXVL-1:128] := 0
VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    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[MAXVL-1:128] := 0
VFNMADD132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[MAXVL-1:128] := 0
VFNMADD213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[MAXVL-1:128] := 0
VFNMADD231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxx__m128 __m128 fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxx__m128 __m128_mask_fnmadd_ss(__m128 a, __m128 b, __m128 c);
VFNMADDxxxx__m128 __m128_maskz_fnmadd_ss(__m128 a, __m128 b, __m128 c);
VFNMADDxxxx__m128 __m128_mask3_fnmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxx__m128 __m128_mask_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxx__m128 __m128_maskz_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxx__m128 __m128_mask3_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxx__m128 __m128_fnmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VE4-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”.
## 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.128.66.0F38.W1 9E/r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 AE/r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 BE/r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 9E/r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 AE/r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 BE/r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 9E/r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 AE/r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 BE/r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 9E/r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 AE/r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 BE/r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 9E/r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 AE/r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 BE/r</td>
<td>B 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>
<td></td>
</tr>
</tbody>
</table>
VFNMSUB132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation
In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
   MAXNUM := 2
ELSEIF (VEX.256)
   MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
   n := 64*i;
   DEST[n+63:n] := RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI
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[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
   ELSE
      SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
   FI
FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask*
      THEN DEST[i+63:i] :=
         RoundFPControl(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
         ELSE ; zeroing-masking
            DEST[i+63:i] := 0
         FI
   FI
ENDFOR
DEST[MAXVL-1:VL] := 0
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[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[MAXVL-1:VL] := 0

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN Dest[i+63:i] :=
    RoundFPControl(-(SRC2[i+63:i]*Dest[i+63:i]) - SRC3[i+63:i])
    ELSE
    IF *merging-masking* ; merging-masking
      THEN *Dest[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      Dest[i+63:i] := 0
    FI
  FI;
ENDFOR
Dest[MAXVL-1:VL] := 0
VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] :=
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
                ELSE
                    DEST[i+63:i] :=
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

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

IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] :=
        RoundFPControl(-SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0
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[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxPD __m512d __m512d_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __m512d_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __m512d_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __m512d_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __m512d_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d __m512d_mask_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __m512d_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m256d __m256d_fnmsub_pd(__m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d __m256d_fnmsub_round_pd(__m256d a, __m256d b, __m256d c, int r);
VFNMSUBxxxPD __m256d __m256d_mask_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d __m256d_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d __m256d_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m128d __m128d_fnmsub_pd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d __m128d_fnmsub_round_pd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxPD __m128d __m128d_mask_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d __m128d_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d __m128d_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, ”Type E2 Class Exception Conditions”.

5-228 Vol. 2C  VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point
### 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.128.66.0F38.w0 9E /r VFNMSUB132PS xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.w0 AE /r VFNMSUB213PS xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.w0 BE /r VFNMSUB231PS xmm1, xmm2, xmm3/m128</td>
<td>A</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.256.66.0F38.w0 9E /r VFNMSUB132PS ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.w0 AE /r VFNMSUB213PS ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.0 BE /r VFNMSUB231PS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply packed single-precision floating-point values from ymm2 and ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w0 9E /r VFNMSUB132PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</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.128.66.0F38.w0 AE /r VFNMSUB213PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</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.128.66.0F38.w0 BE /r VFNMSUB231PS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</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.256.66.0F38.w0 9E /r VFNMSUB132PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</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.256.66.0F38.w0 AE /r VFNMSUB213PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precisions 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.256.66.0F38.w0 BE /r VFNMSUB231PS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</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.512.66.0F38.w0 9E /r VFNMSUB132PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 AE /r VFNMSUB213PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</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 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 BE /r VFNMSUB231PS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result subtract add to zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMz/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMz/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

**VFNMSUB132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**EVEX encoded versions:** The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

**VEX.256 encoded version:** The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register 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 encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

**Operation**

In the operations below, “*” and “-” symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

**VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI

For i = 0 to MAXNUM-1 {
    n := 32*i;
    DEST[n+31:n] := RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}

IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI
VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
 IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
  MAXNUM := 2
ELSEIF (VEX.256)
  MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
 IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
      RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[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
  ELSE
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] :=
    RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
**VFNM.SUB213PS 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
  ELSE
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

**VFNM.SUB231PS 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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR "no writemask"
    THEN DEST[i+31:i] :=
    RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
  ELSE
    IF "merging-masking" ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0


VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

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

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
            DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
        ELSE
            DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
      FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI
      FI
  ENDFOR
  DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxPS __m512 __m512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __m512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __m512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __m512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 __m512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 __m512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 __m512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m256 __m256_fnmsub_ps(__m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __m256_fnmsub_round_ps(__m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m128 __m128_fnmsub_ps(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __m128_fnmsub_round_ps(__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.

INSTRUCTION SET REFERENCE, V-Z
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>64/32bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1, xmm2, xmm3/m64</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1, xmm2, xmm3/m64</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1, xmm2, xmm3/m64</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1 (k1)(z), xmm2, xmm3/m64[er]</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1 (k1)(z), xmm2, xmm3/m64[er]</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1 (k1)(z), xmm2, xmm3/m64[er]</td>
<td>B 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>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</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).

**VFNMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)**

If (EVEX.b = 1) and SRC3 *is a register*

THEN

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
```

ELSE

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
```

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;

```
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```

**VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)**

If (EVEX.b = 1) and SRC3 *is a register*

THEN

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
```

ELSE

```
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
```

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;

```
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0
```
VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF k1[0] or *no writemask*
    THEN
        DEST[63:0] := RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] := 0
        FI;
    FI;
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

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[MAXVL-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[MAXVL-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[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFNMSUBxxSD __m128d _mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxSD __m128d _mm_mask_fnmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxSD __m128d _mm_maskz_fnmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxSD __m128d _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxSD __m128d _mm_mask_fnmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMSUBxxSD __m128d _mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSUBxxSD __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 Table 2-20, "Type 3 Class Exception Conditions".
EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions".
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.LIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1, xmm2, xmm3/m32</td>
<td>A</td>
<td>V/V</td>
<td>FMA</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>VEX.LIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1, xmm2, xmm3/m32</td>
<td>A</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.LIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1, xmm2, xmm3/m32</td>
<td>A</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.LIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1 [k1]{z}, xmm2, xmm3/m32{er}</td>
<td>B</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.LIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1 [k1]{z}, xmm2, xmm3/m32{er}</td>
<td>B</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.LIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1 [k1]{z}, xmm2, xmm3/m32{er}</td>
<td>B</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>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM</td>
<td>reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM</td>
<td>reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM</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.

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

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

ELSE

\[
\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION}(\text{MXCSR.RC});
\]

FI;

IF k1[0] or *no writemask*

THEN \[\text{DEST}[31:0] := \text{RoundFPControl}(- (\text{DEST}[31:0] \times \text{SRC3}[31:0]) - \text{SRC2}[31:0])\] 

ELSE

IF *merging-masking* ; merging-masking

THEN \[\text{DEST}[31:0] \text{ remains unchanged}\]

ELSE ; zeroing-masking

\[\text{DEST}[31:0] := 0\]

FI;

FI;


\[\text{DEST}[\text{MAXVL}-1:128] := 0\]

**VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)**

IF (EVEX.b = 1) and SRC3 *is a register*

THEN

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

ELSE

\[
\text{SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION}(\text{MXCSR.RC});
\]

FI;

IF k1[0] or *no writemask*

THEN \[\text{DEST}[31:0] := \text{RoundFPControl}((- (\text{SRC2}[31:0] \times \text{DEST}[31:0]) - \text{SRC3}[31:0])\] 

ELSE

IF *merging-masking* ; merging-masking

THEN \[\text{DEST}[31:0] \text{ remains unchanged}\]

ELSE ; zeroing-masking

\[\text{DEST}[31:0] := 0\]

FI;

FI;


\[\text{DEST}[\text{MAXVL}-1:128] := 0\]
VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    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[MAXVL-1:128] := 0

VFNMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(-(DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[MAXVL-1:128] := 0

VFNMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(-(SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[MAXVL-1:128] := 0

VFNMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)
DEST[31:0] := RoundFPControl_MXCSR(-(SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[MAXVL-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 __m128_mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 __m128_mm_mask_fnmsub_sc(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 __m128_mm_maskz_fnmsub_sc(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 __m128_mm_fnmsub_round_sc(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 __m128_mm_mask_fnmsub_round_sc(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 __m128_mm_maskz_fnmsub_round_sc(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 __m128_mm_fnmsub_sc (__m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 __m128_mm_fnmsub_sc (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions
VEK-encoded instructions, see Table 2-20, “Type 3 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions”.

5-240  Vol. 2C  VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Val-
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>A</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>A</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>A</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>Tuple Type</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>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The FCLASSPD 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 OR-ed 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 5-4.

![Figure 5-13. Imm8 Byte Specifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS](image)

<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])

    //* 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 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 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:i], imm8[7:0]);
                FI;
            ELSE DEST[j] := 0 ; zeroing-masking only
        FI;
    ENDFOR
DEST[MAX_KL-1:KL] := 0
Intel C/C++ Compiler Intrinsic Equivalent
VFCLASSPD __m128i_mm512_classifier__fpclass_pd_mask(__m512d a, int c);
VFCLASSPD __m128i_mm512_mask_classifier__fpclass_pd_mask(__mmask8 m, __m512d a, int c);
VFCLASSPD __m128i_mm256_classifier__fpclass_pd_mask(__m256d a, int c);
VFCLASSPD __m128i_mm256_mask_classifier__fpclass_pd_mask(__mmask8 m, __m256d a, int c);
VFCLASSPD __m128i_mm_classifier__fpclass_pd_mask(__m128d a, int c);
VFCLASSPD __m128i_mm_mask_classifier__fpclass_pd_mask(__mmask8 m, __m128d a, int c);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Table 2-49, "Type E4 Class Exception Conditions"; additionally:
#UD If EVEX.vvvv != 1111B.
### VFPCLASSPS—Tests Types Of a 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 66 / r ib VFPCLASSPS k2 [k1], xmm2/m128/m32bcst, imm8</td>
<td>A</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.W0 66 / r ib VFPCLASSPS k2 [k1], ymm2/m256/m32bcst, imm8</td>
<td>A</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.W0 66 / r ib VFPCLASSPS k2 [k1], zmm2/m512/m32bcst, imm8</td>
<td>A</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>Tuple Type</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>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

The FCLASSPS 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 5-4.

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

```c
CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    /* Start checking the source operand for special type */
    NegNum := tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
        ELSIF (tsrc[22:0]=0h) Then
            MantAllZeros := 1;
        FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];
    sNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0

```
Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
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() */

VFCLASSPS (EVEX encoded versions)

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

FOR j := 0 TO KL-1
i := j * 32
IF k1[j] OR *no writemask*
   THEN
      IF (EVEX.b == 1) AND (SRC *is memory*)
         THEN
            DEST[j] := CheckFPClassDP(SRC1[31:0], imm8[7:0]);
         ELSE
            DEST[j] := CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
      FI;
   ELSE DEST[j] := 0 ; zeroing-masking only
   FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VFCLASSPS __mmask16 _mm512_fpclass_ps_mask( __m512 a, int c);
VFCLASSPS __mmask16 _mm512_mask_fpclass_ps_mask( __mmask16 m, __m512 a, int c)
VFCLASSPS __mmask8  _mm256_fpclass_ps_mask( __m256 a, int c)
VFCLASSPS __mmask8  _mm256_mask_fpclass_ps_mask( __mmask8 m, __m256 a, int c)
VFCLASSPS __mmask8  _mm_fpclass_ps_mask( __m128 a, int c)
VFCLASSPS __mmask8  _mm_mask_fpclass_ps_mask( __mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions”; additionally:

#UD      If EVEX.vvvv != 1111B.
The FPCLASSSSD 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 5-4.

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];
  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
    FL;
DEST[MAX_KL-1:1] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFPCLASSSD __mmask8 __mm_fpclass_sd_mask(__m128d a, int c)
VFPCLASSSD __mmask8 __mm_mask_fpclass_sd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None

Other Exceptions
See Table 2-53, "Type E6 Class Exception Conditions"; additionally:
#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.LLIG.66.0F3A.W0 67 /r</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The FCLASSSSS 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 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 5-4.

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; F;
    IF (tsrc[30:23]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[22:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    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;

5-248 Vol. 2C VFPCLASSSS—Tests Types Of a Scalar Float32 Values
VFPClssss (EVEX encoded version)
IF k1[0] OR "no writemask"
    THEN DEST[0] :=
        CheckFPClassSP(SRC1[31:0], imm8[7:0])
    ELSE DEST[0] := 0 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFPClssss __mmask8 __mm_fpclass_ss_mask(__m128 a, int c)
VFPClssss __mmask8 __mm_mask_fpclass_ss_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions
None

Other Exceptions
See Table 2-53, "Type E6 Class Exception Conditions"; additionally:
#UD If EVEX.vvvv != 1111B.
VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices

### Instruction Set

#### 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. 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.

### Instruction Encoding

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 2-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.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.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.256.66.0F38.W1 92 /r VGATHERDPD ymm1, vm32x, ymm2</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 ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.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>

---

5-250 Vol. 2C VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices
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)**

MASK[MAXVL-1:128] := 0;
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

DEST[MAXVL-1:128] := 0;

**VGATHERQPD (VEX.128 version)**

MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
  i := j * 64;
  IF MASK[63+i] THEN
    MASK[i +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +63:i] := 0;
  FI;
ENDFOR

FOR j := 0 to 1
  i := j * 64;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63+i] THEN
    DEST[i +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
  FI;
  MASK[i +63: i] := 0;
ENDFOR

DEST[MAXVL-1:128] := 0;
VGATHERQPD (VEX.256 version)

Mask[MAXVL-1:256] := 0;
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

DEST[MAXVL-1:256] := 0;

VGATHERDPD (VEX.256 version)

Mask[MAXVL-1:256] := 0;
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

DEST[MAXVL-1:256] := 0;
**Intel 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 Table 2-27, “Type 12 Class Exception Conditions”.
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.128.66.0F38.W0 92 /r VGATHERDPS xmm1, vm32x, xmm2</td>
<td>A</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.128.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64x, xmm2</td>
<td>A</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.256.66.0F38.W0 92 /r VGATHERDPS ymm1, vm32y, ymm2</td>
<td>A</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.256.66.0F38.W0 93 /r VGATHERQPS xmm1, vm64y, xmm2</td>
<td>A</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 zero 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 zero 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)
MASK[MAXVL-1:128] := 0;
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
FOR j := 0 to 3
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;

VGATHERQPS (VEX.128 version)
MASK[MAXVL-1:64] := 0;
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
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
DEST[MAXVL-1:64] := 0;
VGATHERDPS (VEX.256 version)
MASK[MAXVL-1:256] := 0;
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
DEST[MAXVL-1:256] := 0;

VGATHERQPS (VEX.256 version)
MASK[MAXVL-1:128] := 0;
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
DEST[MAXVL-1:128] := 0;
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 Table 2-27, "Type 12 Class Exception Conditions".
**VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed 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.128.66.0F38.W0 92 /vsib VGATHERDPS xmm1 {k1}, vm32x</td>
<td>A</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>A</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>A</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>A</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>A</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>A</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>Tuple Type</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>Tuple1 Scalar</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 or 4 byte displacement

VGATHERDPS (EVEX encoded version)

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

FOR \(j := 0\) TO \(KL-1\)
  \(i := j \times 32\)
  IF \(k1[j]\)
    THEN \(DEST[i+31:i] := MEM[BASE_ADDR + SignExtend(VINDEX[i+31:i]) \times SCALE + DISP]\)
    \(k1[j] := 0\)
  ELSE \(*DEST[i+31:i] := remains unchanged*\)
  FI;
ENDFOR

\(k1[MAX_KL-1:KL] := 0\)
\(DEST[MAXVL-1:VL] := 0\)

VGATHERDPD (EVEX encoded version)

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

FOR \(j := 0\) TO \(KL-1\)
  \(i := j \times 64\)
  \(k := j \times 32\)
  IF \(k1[j]\)
    THEN \(DEST[i+63:i] := MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) \times SCALE + DISP]\)
    \(k1[j] := 0\)
  ELSE \(*DEST[i+63:i] := remains unchanged*\)
  FI;
ENDFOR

\(k1[MAX_KL-1:KL] := 0\)
\(DEST[MAXVL-1:VL] := 0\)
Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD __m512d __mm512_i32gather_pd(__m256i vdx, void * base, int scale);
VGATHERDPD __m512d __mm512_mask_i32gather_pd(__m512d 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 __mm128_mmask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_i32gather_ps(__m512i vdx, void * base, int scale);
VGATHERDPS __m512 __mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 __mm256_mmask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128 __mm128_mmask_i32gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-61, “Type E12 Class Exception Conditions”.
VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed 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>EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64y</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 93 /vsib VGATHERQPD xmm1 {k1}, vm64x</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</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 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 or 4 byte displacement

**VGATHERQPS (EVEX encoded version)**

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

FOR j := 0 TO KL-1

\[ i := j \times 32 \]
\[ k := j \times 64 \]

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[MAXVL-1:VL/2] := 0

**VGATHERQPD (EVEX encoded version)**

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

FOR j := 0 TO KL-1

\[ i := j \times 64 \]

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[MAXVL-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(__m256d s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 _mm256_mask_i64gather_ps(__m128d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 _mm_mask_i64gather_ps(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 2-61, "Type E12 Class Exception Conditions".
**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 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 42 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</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>VGETEXPPD xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 42 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</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>VGETEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 42 /r</td>
<td>A</td>
<td>V/V</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 under writemask k1.</td>
</tr>
<tr>
<td>VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</td>
<td>ModRMreg (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 5-5.

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 5-5. 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>QNaN(src1)</td>
<td>If (SRC = SNaN) then #IE</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
        {
            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
    }
}
VGETEXPDP (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* ; zeroing-masking
          DEST[i+63:i] := 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VGETEXPDP __m512d _mm512_getexp_pd(__m512d a);
VGETEXPDP __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPDP __m512d _mm512_maskz_getexp_pd( __mmask8 k, __m512d a);
VGETEXPDP __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPDP __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPDP __m512d _mm512_maskz_getexp_round_pd( __mmask8 k, __m512d a, int sae);
VGETEXPDP __m256d _mm256_getexp_pd(__m256d a);
VGETEXPDP __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPDP __m256d _mm256_maskz_getexp_pd( __mmask8 k, __m256d a);
VGETEXPDP __m128d _mm_getexp_pd(__m128d a);
VGETEXPDP __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPDP __m128d _mm_maskz_getexp_pd( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See Table 2-46, “Type E2 Class Exception Conditions”; additionally:
#UD If EVEX.vvvv != 1111B.
VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values

Description

Extracts the biased exponents from the normalized SP FP representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision FP value and written to the corresponding dword elements of the destination operand (the first operand) as SP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

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

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-6.

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.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Table 5-6. 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>If (SRC = SNaN) then #IE</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.

The diagram shows the input and output values for VGETEXPPS when applied to a normalized input. The table below illustrates the process:

**Table 5-14. VGETEXPPS Functionality On Normal Input values**

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRC = 2^1</td>
<td>0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>SAR SRC, 23 = 080h</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>Bias</td>
<td>1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>Tmp - Bias = 1</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>Cx_P2PS(0) = 2^0</td>
<td>0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

**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] := NormalizeExpTinySPFP(SRC[31:0]); // Get Normalized Exponent
    Set #DE
}
ELSE // exponent value is correct
{
    TMP[31:0] := (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction);
}
TMP := SAR(TMP, 23); // Shift Arithmetic Right
TMP := TMP – 127; // Subtract Bias
Return CvtI2S(TMP); // Convert INT to Single-Precision FP number

VGETEXPPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN
                DEST[i+31:i] :=
                ConvertExpSPFP(SRC[31:0])
            ELSE
                DEST[i+31:i] :=
                ConvertExpSPFP(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
            DEST[i+31:i] := 0
            FI
        FI
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPS __m512 __mm512_getexp_ps(__m512 a);
VGETEXPPS __m512 __mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a);
VGETEXPPS __m512 __mm512_maskz_getexp_ps( __mmask16 k, __m512 a);
VGETEXPPS __m512 __mm512_getexp_round_ps( __m512 a, int sae);
VGETEXPPS __m512 __mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae);
VGETEXPPS __m512 __mm512_maskz_getexp_round_ps( __mmask16 k, __m512 a, int sae);
VGETEXPPS __m256 __mm256_getexp_ps(__m256 a);
VGETEXPPS __m256 __mm256_mask_getexp_ps(__m256 s, __mmask8 k, __m256 a);
VGETEXPPS __m256 __mm256_maskz_getexp_ps( __mmask8 k, __m256 a);
VGETEXPPS __m128 __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 Table 2-46, "Type E2 Class Exception Conditions"; additionally:
#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.LLIG.66.0F38.W1 43 /r</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</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 DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location. If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

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 5-5.

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] :=

\[ \text{ConvertExpDPFP(SRC2[63:0])} \]

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged* 

ELSE ; zeroing-masking

DEST[63:0] := 0

FI

FI;

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETEXP SD __m128d __m128d _mm_getexp_sd( __m128d a, __m128d b);
VGETEXP SD __m128d __m128d _mm_mask_getexp_sd( __m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXP SD __m128d __m128d _mm_maskz_getexp_sd( __mmask8 k, __m128d a, __m128d b);
VGETEXP SD __m128d __m128d _mm_getexp_round_sd( __m128d a, __m128d b, int sae);
VGETEXP SD __m128d __m128d _mm_mask_getexp_round_sd( __m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXP SD __m128d __m128d _mm_maskz_getexp_round_sd( __mmask8 k, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions”. 
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.LLG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}[z], xmm2, xmm3/m32{sa}</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (r)</td>
<td>ModRM:r/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.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.

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 5-6.

The formula is:

\[
\text{GETEXP}(x) = \text{floor}(\log_2(|x|))
\]

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

**Operation**

// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPSS
// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPSS

**VGETEXPSS (EVEX encoded version)**

IF k1[0] OR *no writemask*
   THEN DEST[31:0] := ConvertExpDPFP(SRC2[31:0])
ELSE
   IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
   ELSE ; zeroing-masking
      DEST[31:0] := 0
   FI
FI;

DEST[MAXVL-1:128] := 0
Intel C/C++ Compiler Intrinsic Equivalent
VGETEXPSS __m128 _mm_getexp_ss( __m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss( __m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss( __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss( __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss( __m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss( __mmask8 k, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions”. 
VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 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.128.66.0F3A.W1 26 /r ib VGETMANTPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8</td>
<td>A</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>A</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>A</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>Tuple Type</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>Full</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.

For each input DP FP value \( x \), The conversion operation is:

\[
GetMant(x) = \pm 2^k \cdot |x.\text{significand}|
\]

where:

\[
1 \leq |x.\text{significand}| < 2
\]

Unbiased exponent \( k \) can be either 0 or -1, depending on the interval range defined by interv, the range of the significand 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. 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 \( \text{interv} \).

The \text{GetMant()} function follows Table 5-7 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register \( k1 \) are computed and stored into the destination. Elements in \( \text{zmm1} \) with the corresponding bit clear in \( k1 \) retain their previous values.

Note: \text{EVEX.vvvv} is reserved and must be \text{1111b}; otherwise instructions will \#UD.

### Table 5-7. \text{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 ( \text{interv} )</td>
</tr>
<tr>
<td>+\infty</td>
<td>1.0</td>
<td>Ignore ( \text{interv} )</td>
</tr>
<tr>
<td>0</td>
<td>1.0</td>
<td>Ignore ( \text{interv} )</td>
</tr>
<tr>
<td>-0</td>
<td>IF (SC[0]) THEN +1.0 ELSE -1.0</td>
<td>Ignore ( \text{interv} )</td>
</tr>
<tr>
<td>-\infty</td>
<td>IF (SC[1]) THEN QNaN_Indefinite} ELSE { IF (SC[0]) THEN +1.0 ELSE -1.0</td>
<td>Ignore ( \text{interv} ) If (SC[1]) then #IE</td>
</tr>
<tr>
<td>negative</td>
<td>SC[1]? QNaN_Indefinite : Getmant(SRC)(^1)</td>
<td>If (SC[1]) then #IE</td>
</tr>
</tbody>
</table>

**NOTES:**
1. In case \( SC[1] = 0 \), the sign of Getmant(SRC) is declared according to \( SC[0] \).

### Operation

```python
def getmant_fp64(src, sign_control, normalization_interval):
    bias := 1023
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
    denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
    infinity := (dst.exp = 0x7FF) and (dst.fraction = 0)
    nan := (dst.exp = 0x7FF) and (dst.fraction != 0)
    src_signaling := src.fraction[51]
    snan := nan and (src_signaling = 0)
    positive := (src.sign = 0)
    negative := (src.sign = 1)
    if nan:
        if snan:
            MXCSR.IE := 1
            return qnan(src)
        if positive and (zero or infinity):
            return 1.0
        if negative:
            if zero:
                return signed_one
            if infinity:
```
if sign_control[1]:
    MXCSR.IE := 1
    return QNaN_Indefinite
return signed_one

if denormal:
    jbit := 0
    dst.exp := bias
    while jbit = 0:
        jbit := dst.fraction[51]
        dst.fraction := dst.fraction << 1
        dst.exp := dst.exp - 1
    MXCSR.DE := 1

    unbiased_exp := dst.exp - bias
    odd_exp := unbiased_exp[0]
    signaling_bit := dst.fraction[51]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:
        dst.exp := bias-1
    else if normalization_interval = 0b11:
        dst.exp := signaling_bit ? bias-1 : bias
    return dst

VGETMANTPD (EVEX encoded versions)
VGETMANTPD dest[k1], src, imm8
VL = 128, 256, or 512
KL := VL / 64
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.double[0]
        ELSE:
            tsrc := src.double[i]
        DEST.double[i] := getmant_fp64(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.double[i] := 0
        //else DEST.double[i] remains unchanged
    DEST[MAX_VL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VGETMANTPD __m512d _mm512_getmant_pd(__m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_maskz_getmant_pd(__mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_getmant_round_pd(__m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_maskz_getmant_round_pd(__mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d _mm256_getmant_pd(__m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_maskz_getmant_pd(__mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_getmant_pd(__m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_maskz_getmant_pd(__mmask8 k, __m128d a, enum intv, enum sgn);

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Table 2-46, "Type E2 Class Exception Conditions"; additionally:

```
#UD If EVEX.vvvv != 1111B.
```
VGETMANTPS—Extract Float32 Vector of Normalized Mantissas 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.128.66.0F3A.W0 26 /r ib VGETMANTPS xmm1 [k1]{z}, xmm2/m128/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Get normalized mantissa from float32 vector xmm2/m128/m32bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 26 /r ib VGETMANTPS ymm1 [k1]{z}, ymm2/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Get normalized mantissa from float32 vector ymm2/m256/m32bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 26 /r ib VGETMANTPS zmm1 [k1]{z}, zmm2/m512/m32bcst{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Get normalized mantissa from float32 vector zmm2/m512/m32bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
</tbody>
</table>

**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 can be either 0 or -1, depending on the interval range defined by interv, the range of the significand 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. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-7 when dealing with floating-point special numbers.

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

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.
Operation

def getmant_fp32(src, sign_control, normalization_interval):
    bias := 127
    dst.sign := sign_control[0] ? 0 : src.sign
    signed_one := sign_control[0] ? +1.0 : -1.0
    dst.exp := src.exp
    dst.fraction := src.fraction
    zero := (dst.exp = 0) and ((dst.fraction = 0) or (MXCSR.DAZ=1))
denormal := (dst.exp = 0) and (dst.fraction != 0) and (MXCSR.DAZ=0)
infinity := (dst.exp = 0xFF) and (dst.fraction = 0)
nan := (dst.exp = 0xFF) and (dst.fraction != 0)
src_signaling := src.fraction[22]

    if nan:
        if snan:
            MXCSR.IE := 1
        return qnan(src)
    if positive and (zero or infinity):
        return 1.0
    if negative:
        if zero:
            return signed_one
        if infinity:
            if sign_control[1]:
                MXCSR.IE := 1
                return QNaN_Indefinite
            return signed_one
        if sign_control[1]:
            MXCSR.IE := 1
            return QNaN_Indefinite
    if denormal:
        jbit := 0
        dst.exp := bias
        while jbit = 0:
            jbit := dst.fraction[22]
            dst.fraction := dst.fraction << 1
            dst.exp := dst.exp - 1
            MXCSR.DE := 1

    unbiased_exp := dst.exp - bias
    odd_exp := unbiased_exp[0]
    signaling_bit := dst.fraction[22]
    if normalization_interval = 0b00:
        dst.exp := bias
    else if normalization_interval = 0b01:
        dst.exp := odd_exp ? bias-1 : bias
    else if normalization_interval = 0b10:
        dst.exp := bias-1
    else if normalization_interval = 0b11:
        dst.exp := signaling_bit ? bias-1 : bias
VGETMANTPS (EVEX encoded versions)
VGETMANTPS dest{k1}, src, imm8
VL = 128, 256, or 512
KL := VL / 32
sign_control := imm8[3:2]
normalization_interval := imm8[1:0]

FOR i := 0 to KL-1:
    IF k1[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := src.float[0]
        ELSE:
            tsrc := src.float[i]
        DEST.float[i] := getmant_fp32(tsrc, sign_control, normalization_interval)
    ELSE IF *zeroing*:
        DEST.float[i] := 0
        //else DEST.float[i] remains unchanged

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 __mm128_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 Table 2-46, "Type E2 Class Exception Conditions"; additionally:

#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.LLIG.66.0F3A.W1 27 /r ib</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</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 can be either 0 or -1, depending on the interval range defined by interv, the range of the significand 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. 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 5-7 when dealing with floating-point special numbers.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.
Operation
// getmant_fp64(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPD

VGETMANTSD (EVEX encoded version)
SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
  THEN DEST[63:0] :=
    getmant_fp64(src, sign_control, normalization_interval)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
      ELSE ; zeroing-masking
        DEST[63:0] := 0
    FI
FI
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VGETMANTSD _m128d __m_getmant_sd( _m128d a, _m128d b, enum intv, enum sgn);
VGETMANTSD _m128d __m_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD _m128d __m_maskz_getmant_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD _m128d __m_getmant_round_sd( __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD _m128d __m_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD _m128d __m_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 Table 2-47, "Type E3 Class Exception Conditions".
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.LLIG.66.0F3A.W0 27 /r ib</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:rr/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:

\[
GetMant(x) = \pm 2^{k}|x.\text{significand}|
\]

where:

\[1 \leq |x.\text{significand}| < 2\]

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand 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. 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 5-7 when dealing with floating-point special numbers.

If writemasking is used, the low doubleword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low doubleword element of the destination operand is unconditionally updated.
**Operation**

// getmant_fp32(src, sign_control, normalization_interval) is defined in the operation section of VGETMANTPS

**VGETMANTSS (EVEX encoded version)**

SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
        getmant_fp32(src, sign_control, normalization_interval)
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] := 0
        FI
    FI;
DEST[MAXVL-1:128] := 0

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

VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Table 2-47, "Type E3 Class Exception Conditions".
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.256.66.0F3A.W0 18 / r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 18 / r ib VINSERTF32X4 ymm1 [k1][z], ymm2, xmm3/m128, imm8</td>
<td>C 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 18 / r ib VINSERTF32X4 zmm1 [k1][z], zmm2, xmm3/m128, imm8</td>
<td>C 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 18 / r ib VINSERTF64X2 ymm1 [k1][z], ymm2, xmm3/m128, imm8</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 18 / r ib VINSERTF64X2 zmm1 [k1][z], zmm2, xmm3/m128, imm8</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1A / r ib VINSERTF32X8 zmm1 [k1][z], zmm2, xmm3/m256, imm8</td>
<td>D 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 1A / r ib VINSERTF64X4 zmm1 [k1][z], zmm2, ymm3/m256, imm8</td>
<td>C 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>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>B</td>
<td>Tuple2</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>C</td>
<td>Tuple4</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
<tr>
<td>D</td>
<td>Tuple8</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</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]
ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF
  00: TEMP_DEST[127:0] := 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[MAXVL-1:VL] := 0

**VINSERTF64x2 (EVEX encoded versions)**

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

TEMP_DEST[VL-1:0] := SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF
  0: TEMP_DEST[127:0] := SRC2[127:0]
ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF
  00: TEMP_DEST[127:0] := 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] := 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] := 0
IF *merging-masking*; merging-masking
    THEN *DEST[i+63:j] remains unchanged*
ELSE ; zeroing-masking
    DEST[i+63:i] := 0
FI

ENDFOR
DEST[MAXVL-1:VL] := 0

**VINSERTF32x8 (EVEX.U1.512 encoded version)**

TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0

**VINSERTF64x4 (EVEX.512 encoded version)**

VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking*; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
VINSERTF128 (VEX encoded version)
TEMP[255:0] := SRC1[255:0]
CASE (imm8[0]) OF
  0: TEMP[127:0] := SRC2[127:0]
ESAC
DEST := TEMP

Intel C/C++ Compiler Intrinsic Equivalent
VINSERTF32x4 __m512 _mm512_insertf32x4(__m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_mask_insertf32x4(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_maskz_insertf32x4(__mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_insertf32x4(__m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_mask_insertf32x4(__m256 s, __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_maskz_insertf32x4(__mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x8 __m512 _mm512_insertf32x8(__m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_mask_insertf32x8(__m512 s, __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_maskz_insertf32x8(__mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF64x2 __m512d _mm512d_insertf64x2(__m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512d_mask_insertf64x2(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512d_maskz_insertf64x2(__mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256d_insertf64x2(__m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256d_mask_insertf64x2(__m256d s, __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256d_maskz_insertf64x2(__mmask8 k, __m256d a, __m128d b, int imm);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instruction, see Table 2-23, “Type 6 Class Exception Conditions”; additionally:
#UD If VEX.L = 0.
EVEX-encoded instruction, see Table 2-54, "Type E6NF Class Exception Conditions".
VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

**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: TEMP_DEST[127:0] := SRC2[127:0]
ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF
  00: TEMP_DEST[127:0] := 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[MAXVL-1:VL] := 0

**VINSERTI64x2 (EVEX encoded versions)**

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

TEMP_DEST[VL-1:0] := SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF
  0: TEMP_DEST[127:0] := SRC2[127:0]
ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF
  00: TEMP_DEST[127:0] := 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] := 0
  FI
  FI;

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

IF
ENDFOR
DEST[MAXVL-1:VL] := 0

**VINSERTI32x8 (EVEX.U1.512 encoded version)**
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VINSERTI64x4 (EVEX.512 encoded version)**
VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
**VINSERTI128**

\[ \text{TEMP}[255:0] := \text{SRC1}[255:0] \]

CASE (imm8[0]) OF
  0: \[ \text{TEMP}[127:0] := \text{SRC2}[127:0] \]
  1: \[ \text{TEMP}[255:128] := \text{SRC2}[127:0] \]
ESAC

DEST := TEMP

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

\[
\begin{align*}
\text{VINSERTI32x4} & \quad \text{mm512i } _\text{inserti32x4}(\text{m512i} a, \text{m128i} b, \text{int} \text{imm}); \\
\text{VINSERTI32x4} & \quad \text{mm512i } _\text{mask_inserti32x4}(\text{m512i} s, \text{mmask16} k, \text{m512i} a, \text{m128i} b, \text{int} \text{imm}); \\
\text{VINSERTI32x4} & \quad \text{mm512i } _\text{maskz_inserti32x4}(\text{mmask16} k, \text{m512i} a, \text{m128i} b, \text{int} \text{imm}); \\
\text{VINSERTI32x4} & \quad \text{m256i } _\text{inserti32x4}(\text{m256i} a, \text{m128i} b, \text{int} \text{imm}); \\
\text{VINSERTI32x4} & \quad \text{m256i } _\text{mask_inserti32x4}(\text{m256i} s, \text{mmask8} k, \text{m256i} a, \text{m128i} b, \text{int} \text{imm}); \\
\text{VINSERTI32x4} & \quad \text{m256i } _\text{maskz_inserti32x4}(\text{mmask8} k, \text{m256i} a, \text{m128i} b, \text{int} \text{imm}); \\
\text{VINSERTI32x4} & \quad \text{mm512i } _\text{inserti32x8}(\text{m512i} a, \text{m256i} b, \text{int} \text{imm}); \\
\text{VINSERTI32x4} & \quad \text{mm512i } _\text{mask_inserti32x8}(\text{m512i} s, \text{mmask16} k, \text{m512i} a, \text{m256i} b, \text{int} \text{imm}); \\
\text{VINSERTI32x4} & \quad \text{mm512i } _\text{maskz_inserti32x8}(\text{mmask16} k, \text{m512i} a, \text{m256i} b, \text{int} \text{imm}); \\
\text{VINSERTI64x2} & \quad \text{mm512i } _\text{inserti64x2}(\text{m512i} a, \text{m128i} b, \text{int} \text{imm}); \\
\text{VINSERTI64x2} & \quad \text{mm512i } _\text{mask_inserti64x2}(\text{m512i} s, \text{mmask8} k, \text{m512i} a, \text{m128i} b, \text{int} \text{imm}); \\
\text{VINSERTI64x2} & \quad \text{mm512i } _\text{maskz_inserti64x2}(\text{mmask8} k, \text{m512i} a, \text{m128i} b, \text{int} \text{imm}); \\
\text{VINSERTI64x4} & \quad \text{mm512i } _\text{inserti64x4}(\text{m512i} a, \text{m256i} b, \text{int} \text{imm}); \\
\text{VINSERTI64x4} & \quad \text{mm512i } _\text{mask_inserti64x4}(\text{m512i} s, \text{mmask8} k, \text{m512i} a, \text{m256i} b, \text{int} \text{imm}); \\
\text{VINSERTI64x4} & \quad \text{mm512i } _\text{maskz_inserti64x4}(\text{mmask8} k, \text{m512i} a, \text{m256i} b, \text{int} \text{imm}); \\
\text{VINSERTI28} & \quad \text{mm256i } _\text{insertf128_si256}(\text{m256i} a, \text{m128i} b, \text{int} \text{offset});
\end{align*}
\]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VEX-encoded instruction, see Table 2-23, “Type 6 Class Exception Conditions”; additionally:

\#UD  \quad \text{If VEX.L = 0.}

EVEX-encoded instruction, see Table 2-54, “Type E6NF Class Exception Conditions”. 

---

VINSETRI128/VINSETRI32x4/VINSETRI64x2/VINSETRI32x8/VINSETRI64x4—Insert Packed Integer Values
VMASKMOV—Conditional SIMD 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.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.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.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.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.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.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.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.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>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 (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>MVR</td>
<td>ModRMreg/m (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (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[MAXVL-1:128] := 0

**VMASKMOVPS - 256-bit load**

DEST[31:0] := IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] := IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] := IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] := IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] := IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] := IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] := IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] := IF (SRC1[255]) Load_32(mem + 28) ELSE 0

**VMASKMOVPD - 128-bit load**

DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[MAXVL-1:128] := 0

**VMASKMOVPD - 256-bit load**

DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[195:128] := IF (SRC1[191]) Load_64(mem + 24) ELSE 0
DEST[255:196] := IF (SRC1[255]) Load_64(mem + 28) ELSE 0

**VMASKMOVPS - 128-bit store**

IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]

**VMASKMOVPS - 256-bit store**

IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]
IF (SRC1[159]) DEST[159:128] := SRC2[159:128]
VMASKMOV - 128-bit store
IF (SRC[63]) DEST[63:0] := SRC2[63:0]
IF (SRC[127]) DEST[127:64] := SRC2[127:64]

VMASKMOV - 256-bit store
IF (SRC[63]) DEST[63:0] := SRC2[63:0]
IF (SRC[127]) DEST[127:64] := SRC2[127:64]

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  _mm_maskload_ps(float const *a, __m128i mask)
void    _mm_maskstore_ps(float *a, __m128i mask, __m128 b)
__m128d _mm_maskload_pd(double const *a, __m128i mask);
void    _mm_maskstore_pd(double *a, __m128i mask, __m128d b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Table 2-23, "Type 6 Class Exception Conditions" (No AC# reported for any mask bit combinations); additionally:
#UD            If VEX.W = 1.
VP2INTERSECTD/VP2INTERSECTQ—Compute Intersection Between DWORDS/QUADWORDS to a Pair of Mask Registers

<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.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in xmm3/m128/m32bcst and xmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in ymm3/m256/m32bcst and ymm2.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F2.0F38.W0 68 /r VP2INTERSECTD k1+1, zmm2, zmm3/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between dwords in zmm3/m512/m32bcst and zmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.128.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, xmm2, xmm3/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in xmm3/m128/m64bcst and xmm2.</td>
</tr>
<tr>
<td>EVEX.NDS.256.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, ymm2, ymm3/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in ymm3/m256/m64bcst and ymm2.</td>
</tr>
<tr>
<td>EVEX.NDS.512.F2.0F38.W1 68 /r VP2INTERSECTQ k1+1, zmm2, zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512_VP2INTERSECT</td>
<td>Store, in an even/odd pair of mask registers, the indicators of the locations of value matches between quadwords in zmm3/m512/m64bcst and zmm2.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Full</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 writes an even/odd pair of mask registers. The mask register destination indicated in the MODRM.REG field is used to form the basis of the register pair. The low bit of that field is masked off (set to zero) to create the first register of the pair.

EVEX.aaa and EVEX.z must be zero.
Operation

VP2INTERSECTD destmask, src1, src2
(KL, VL) = (4, 128), (8, 256), (16, 512)

// dest_mask_reg_id is the register id specified in the instruction for destmask
dest_base := dest_mask_reg_id & ~1

// maskregs[ ] is an array representing the mask registers
maskregs[dest_base+0][MAX_KL-1:0] := 0
maskregs[dest_base+1][MAX_KL-1:0] := 0

FOR i := 0 to KL-1:
    FOR j := 0 to KL-1:
        match := (src1.dword[i] == src2.dword[j])
        maskregs[dest_base+0].bit[i] |= match
        maskregs[dest_base+1].bit[j] |= match

VP2INTERSECTQ destmask, src1, src2
(KL, VL) = (2, 128), (4, 256), (8, 512)

// dest_mask_reg_id is the register id specified in the instruction for destmask
dest_base := dest_mask_reg_id & ~1

// maskregs[ ] is an array representing the mask registers
maskregs[dest_base+0][MAX_KL-1:0] := 0
maskregs[dest_base+1][MAX_KL-1:0] := 0

FOR i = 0 to KL-1:
    FOR j = 0 to KL-1:
        match := (src1.qword[i] == src2.qword[j])
        maskregs[dest_base+0].bit[i] |= match
        maskregs[dest_base+1].bit[j] |= match

Intel C/C++ Compiler Intrinsic Equivalent

VP2INTERSECTD void _mm_2intersect_epi32(__m128i, __m128i, __mmask8 *, __mmask8 *);
VP2INTERSECTD void _mm256_2intersect_epi32(__m256i, __m256i, __mmask8 *, __mmask8 *);
VP2INTERSECTQ void _mm_2intersect_epi64(__m128i, __m128i, __mmask8 *, __mmask8 *);
VP2INTERSECTQ void _mm256_2intersect_epi64(__m256i, __m256i, __mmask8 *, __mmask8 *);
VP2INTERSECTQ void _mm512_2intersect_epi64(__m512i, __m512i, __mmask8 *, __mmask8 *);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-50, “Type E4NF Class Exception Conditions”.

INSTRUCTION SET REFERENCE, V-Z
VPBLENDD — Blend Packed Dwords

### 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 (r)</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 (MAXVL-1:128) of the corresponding YMM register are zeroed.

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

### Operation

**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]
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[MAXVL-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 Table 2-21, “Type 4 Class Exception Conditions”; 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.128.66.0F38.W0 66 /r</td>
<td>A</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>VPBLENDMB xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 66 /r</td>
<td>A</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>VPBLENDMB ymm1 (k1)[z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 66 /r</td>
<td>A</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>VPBLENDMB zmm1 (k1)[z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 66 /r</td>
<td>A</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>VPBLENDMW xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 66 /r</td>
<td>A</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>VPBLENDMW ymm1 (k1)[z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 66 /r</td>
<td>A</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>
<tr>
<td>VPBLENDMW zmm1 (k1)[z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full Mem</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 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 \(KL - 1\)

\[i := j \times 8\]

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

THEN \(DEST[i+7:i] := SRC2[i+7:i]\)

ELSE

IF *merging-masking* ; merging-masking

THEN \(DEST[i+7:i] := SRC1[i+7:i]\)

ELSE ; zeroing-masking

\(DEST[i+7:i] := 0\)

FI;

FI;

ENDFOR

\(DEST[MAXVL-1:VL] := 0;\)

**VPBLENDMW (EVEX encoded versions)**

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

FOR \(j := 0\) TO \(KL - 1\)

\[i := j \times 16\]

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

THEN \(DEST[i+15:i] := SRC2[i+15:i]\)

ELSE

IF *merging-masking* ; merging-masking

THEN \(DEST[i+15:i] := SRC1[i+15:i]\)

ELSE ; zeroing-masking

\(DEST[i+15:i] := 0\)

FI;

FI;

ENDFOR

\(DEST[MAXVL-1:VL] := 0;\)

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

**VPBLENDMB**

\[
\begin{align*}
\text{__m512i } & \text{ _mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);} \\
\text{__m256i } & \text{ _mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);} \\
\text{__m128i } & \text{ _mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);} \\
\end{align*}
\]

**VPBLENDMW**

\[
\begin{align*}
\text{__m512i } & \text{ _mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);} \\
\text{__m256i } & \text{ _mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);} \\
\text{__m128i } & \text{ _mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);} \\
\end{align*}
\]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions".
VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 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.128.66.0F38.W0 64 /r VPBLENDMD xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>A</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.256.66.0F38.W0 64 /r VPBLENDMD ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>A</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.512.66.0F38.W0 64 /r VPBLENDMD zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 64 /r VPBLENDMQ xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>A</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.256.66.0F38.W1 64 /r VPBLENDMQ ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>A</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.512.66.0F38.W1 64 /r VPBLENDMQ zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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 an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

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

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

VPBLENDMD (EVEX encoded versions)

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

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

DEST[MAXVL-1:VL] := 0;

VPBLENDMD (EVEX encoded versions)

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

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

DEST[MAXVL-1:VL] := 0;
Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMD __m512i _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);
VPBLENDMD __m256i _mm256_mask_blend_epi32(__mmask8 m, __m256i a, __m256i b);
VPBLENDMD __m128i _mm_mask_blend_epi32(__mmask8 m, __m128i a, __m128i b);
VPBLENDMQ __m512i _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);
VPBLENDMQ __m256i _mm256_mask_blend_epi64(__mmask8 m, __m256i a, __m256i b);
VPBLENDMQ __m128i _mm_mask_blend_epi64(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions".
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</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7A /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7A /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7B /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7B /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7B /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7C /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7C /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7C /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 7C /r</td>
<td>A V/N.E.¹</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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7C /r</td>
<td>A V/N.E.¹</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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7C /r</td>
<td>A V/N.E.¹</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>
<td></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>Tuple Type</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>Tuple1 Scalar</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[MAXVL-1:VL] := 0

VPBROADCASTW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
   i := j * 16
   IF k1[j] OR *no writemask*
      THEN DEST[i+15:i] := SRC[15:0]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+15:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+15:i] := 0
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPBROADCASTD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask*
      THEN DEST[i+31:i] := SRC[31:0]
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+31:i] := 0
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPBROADCASTQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := SRC[63:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPBROADCASTB __m512i _mm512_mask_set1_epi8(__m512i s, __mmask64 k, int a);
VPBROADCASTB __m512i _mm512_maskz_set1_epi8(__mmask64 k, int a);
VPBROADCASTB __m256i _mm256_mask_set1_epi8(__m256i s, __mmask32 k, int a);
VPBROADCASTB __m256i _mm256_maskz_set1_epi8(__mmask32 k, int a);
VPBROADCASTB __m128i _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, __mmask4 k, __int64 a);
VPBROADCASTQ __m256i _mm256_maskz_set1_epi64(__mmask4 k, __int64 a);
VPBROADCASTQ __m128i _mm_mask_set1_epi64(__m128i s, __mmask4 k, __int64 a);
VPBROADCASTQ __m128i _mm_maskz_set1_epi64(__mmask4 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, __mmask8 k, __int64 a);
VPBROADCASTW __m256i _mm256_maskz_set1_epi16(__mmask8 k, __int64 a);
VPBROADCASTW __m128i _mm_mask_set1_epi16(__m128i s, __mmask8 k, __int64 a);
VPBROADCASTW __m128i _mm_maskz_set1_epi16(__mmask8 k, __int64 a);

Exceptions
EVEX-encoded instructions, see Table 2-55, ”Type E7NM Class Exception Conditions”; additionally:
#UD     If EVEX.vvvv != 1111B.
### VPBROADCAST—Load Integer and Broadcast

<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 78 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Broadcast a byte integer in the source operand to sixteen locations in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 78 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Broadcast a byte integer in the source operand to thirty-two locations in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 78 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 78 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 78 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 79 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Broadcast a word integer in the source operand to eight locations in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 79 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Broadcast a word integer in the source operand to sixteen locations in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 79 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 79 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 79 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 58 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Broadcast a dword integer in the source operand to four locations in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 58 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Broadcast a dword integer in the source operand to eight locations in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 58 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 58 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 58 /r</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 59 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Broadcast a qword element in source operand to two locations in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 59 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Broadcast a qword element in source operand to four locations in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 59 /r</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 59 /r</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 59 /r</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 59 /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>Opcode/Instruction</td>
<td>Op / En</td>
<td>64/32 bit Mode Support</td>
<td>CPUID Feature Flag</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------</td>
<td>------------------------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 59 /r VBBROADCASTI32x2 ymm1 (k1)[z], xmm2/m64</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 59 /r VBBROADCASTI32x2 zmm1 (k1)[z], xmm2/m64</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 5A /r VBBROADCASTI128 ymm1, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 5A /r VBBROADCASTI32X4 ymm1 (k1)[z], m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512VL, AVX512F</td>
<td>Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 5A /r VBBROADCASTI32X4 zmm1 (k1)[z], m128</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 5A /r VBBROADCASTI64X2 ymm1 (k1)[z], m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL, AVX512DQ</td>
<td>Broadcast 128 bits of 2 quadword integer data in mem to locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 5A /r VBBROADCASTI64X2 zmm1 (k1)[z], m128</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Broadcast 128 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 5B /r VBBROADCASTI32X8 zmm1 (k1)[z], m256</td>
<td>E</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 5B /r VBBROADCASTI64X4 zmm1 (k1)[z], m256</td>
<td>D</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Tuple2</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Tuple4</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>E</td>
<td>Tuple8</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/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 VBBROADCASTB/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. VBBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VBBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

- **EVEX-encoded VBBROADCASTD/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.

- **VBBROADCASTI32X4 and VBBROADCASTI64X4:** 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 VBBROADCASTI32X4 and VBBROADCASTI64X4 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)
Operation

VPBROADCASTB (EVEX encoded versions)

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

FOR \(j := 0\) TO \(KL-1\)

\[i := j \times B\]

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

THEN \(\text{DEST}[i+7:i] := \text{SRC}[7:0]\)

ELSE

IF *merging-masking* ; merging-masking

THEN *\(\text{DEST}[i+7:i]\) remains unchanged* ; zeroing-masking

ELSE

\(\text{DEST}[i+7:i] := 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)
VPBROADCASTW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SRC[15:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+15:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPBROADCASTD (128 bit version)
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[MAXVL-1:128] := 0

VPBROADCASTD (VEX.256 encoded version)
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[159:128] := temp
DEST[191:160] := temp
DEST[223:192] := temp
DEST[255:224] := temp
DEST[MAXVL-1:256] := 0

VPBROADCASTD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SRC[31:0]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPBROADCASTQ (VEX.256 encoded version)
\[
\text{temp} := \text{SRC}[63:0] \\
\text{DEST}[63:0] := \text{temp} \\
\text{DEST}[127:64] := \text{temp} \\
\text{DEST}[191:128] := \text{temp} \\
\text{DEST}[255:192] := \text{temp} \\
\text{DEST}[\text{MAXVL}-1:256] := 0
\]

VPBROADCASTQ (EVEX encoded versions)
\[(KL, VL) = (2, 128), (4, 256), (8, 512)\]
\[
\text{FOR } j := 0 \text{ TO } KL-1 \\
\quad i := j \times 64 \\
\quad \text{IF } k1[j] \text{ OR } \text{no writemask} \\
\quad \quad \text{THEN } \text{DEST}[i+63:i] := \text{SRC}[63:0] \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \text{IF } \text{merging-masking} \\
\quad \quad \quad \quad \text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged} \\
\quad \quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \quad \text{DEST}[i+63:i] := 0 \\
\quad \quad \FI
\]
\[
\text{ENDFOR} \\
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

VBROADCASTI32x2 (EVEX encoded versions)
\[(KL, VL) = (4, 128), (8, 256), (16, 512)\]
\[
\text{FOR } j := 0 \text{ TO } KL-1 \\
\quad i := j \times 32 \\
\quad n := (j \mod 2) \times 32 \\
\quad \text{IF } k1[j] \text{ OR } \text{no writemask} \\
\quad \quad \text{THEN } \text{DEST}[i+31:i] := \text{SRC}[n+31:n] \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \text{IF } \text{merging-masking} \\
\quad \quad \quad \quad \text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged} \\
\quad \quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \quad \text{DEST}[i+31:i] := 0 \\
\quad \quad \FI
\]
\[
\text{ENDFOR} \\
\text{DEST}[\text{MAXVL}-1:VL] := 0
\]

VBROADCASTI128 (VEX.256 encoded version)
\[
\text{temp} := \text{SRC}[127:0] \\
\text{DEST}[127:0] := \text{temp} \\
\text{DEST}[255:128] := \text{temp} \\
\text{DEST}[\text{MAXVL}-1:256] := 0
\]
VBROADCASTI32X4 (EVEX encoded versions)
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    n := (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VBROADCASTI64X2 (EVEX encoded versions)
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 64
    n := (j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := SRC[n+63:n]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
    FI;
ENDFOR;

VBROADCASTI32X8 (EVEX.U1.512 encoded version)
FOR j := 0 TO 15
    i := j * 32
    n := (j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SRC[n+31:n]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VBBROADCASTI64X4 (EVEX.512 encoded version)
FOR j := 0 TO 7
i := j * 64
n := (j modulo 4) * 64
IF k1[j] OR *no writemask*
THEN DEST[i+63:i] := SRC[n+63:n]
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] := 0
FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPBROADCASTB __m512i _mm512_broadcastb_epi8( __m128i a);
VPBROADCASTB __m512i _mm512_mask_broadcastb_epi8( __m128i s, __mmask64 k, __m128i a);
VPBROADCASTB __m512i _mm512_maskz_broadcastb_epi8( __mmask64 k, __m128i a);
VPBROADCASTB __m256i _mm256_broadcastb_epi8( __m128i a);
VPBROADCASTB __m256i _mm256_mask_broadcastb_epi8( __m128i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i _mm256_maskz_broadcastb_epi8( __mmask32 k, __m128i a);
VPBROADCASTB __m128i _mm_broadcastb_epi8( __m128i a);
VPBROADCASTB __m128i _mm_mask_broadcastb_epi8( __m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_maskz_broadcastb_epi8( __mmask16 k, __m128i a);

VPBROADCASTQ __m512i _mm512_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m512i _mm512_mask_broadcastq_epi64( __m128i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m256i _mm256_mask_broadcastq_epi64( __m128i s, __mmask4 k, __m128i a);
VPBROADCASTQ __m256i _mm256_maskz_broadcastq_epi64( __mmask4 k, __m128i a);
VPBROADCASTQ __m128i _mm_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m128i _mm_mask_broadcastq_epi64( __m128i s, __mmask2 k, __m128i a);
VPBROADCASTQ __m128i _mm_maskz_broadcastq_epi64( __mmask2 k, __m128i a);

VPBROADCASTW __m512i _mm512_broadcastw_epi16( __m128i a);
VPBROADCASTW __m512i _mm512_mask_broadcastw_epi16( __m128i s, __mmask1 k, __m128i a);
VPBROADCASTW __m512i _mm512_maskz_broadcastw_epi16( __mmask1 k, __m128i a);
VPBROADCASTW __m256i _mm256_broadcastw_epi16( __m128i a);
VPBROADCASTW __m256i _mm256_mask_broadcastw_epi16( __m128i s, __mmask4 k, __m128i a);
VPBROADCASTW __m256i _mm256_maskz_broadcastw_epi16( __mmask4 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);
VPBROADCASTW __m128i _mm512_broadcastw_i32x2( __m128i a);
VPBROADCAST—Load Integer and Broadcast

INSTRUCTION SET REFERENCE, V-Z

Vol. 2C 5-319

VBROADCASTI32x2 __m512i __mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x2 __m512i __mm512_maskz_broadcast_i32x2(__mmask16 k, __m128i a);
VBROADCASTI32x2 __m256i __mm256_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m256i __mm256_mask_broadcast_i32x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m256i __mm256_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i __mm128_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m128i __mm128_mask_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i __mm128_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x4 __m512i __mm512_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m512i __mm512_mask_broadcast_i32x4(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x4 __m512i __mm512_maskz_broadcast_i32x4(__mmask16 k, __m128i a);
VBROADCASTI32x4 __m256i __mm256_broadcast_i32x4(__m128i a);
VBroadcastI32x4 __m256i __mm256_mask_broadcast_i32x4(__m256i s, __mmask8 k, __m128i a);
VBroadcastI32x4 __m256i __mm256_maskz_broadcast_i32x4(__mmask8 k, __m128i a);
VBROADCASTI32x8 __m512i __mm512_broadcast_i32x8(__m256i a);
VBroadcastI32x8 __m512i __mm512_mask_broadcast_i32x8(__m512i s, __mmask16 k, __m256i a);
VBroadcastI32x8 __m512i __mm512_maskz_broadcast_i32x8(__mmask16 k, __m256i a);
VBROADCASTI64x2 __m512i __mm512_broadcast_i64x2(__m128i a);
VBroadcastI64x2 __m512i __mm512_mask_broadcast_i64x2(__m512i s, __mmask8 k, __m128i a);
VBroadcastI64x2 __m512i __mm512_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBroadcastI64x2 __m256i __mm256_broadcast_i64x2(__m256i s, __mmask8 k, __m128i a);
VBroadcastI64x2 __m256i __mm256_mask_broadcast_i64x2(__m256i s, __mmask8 k, __m128i a);
VBroadcastI64x2 __m256i __mm256_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBroadcastI64x4 __m512i __mm512_broadcast_i64x4(__m512i s, __mmask8 k, __m256i a);
VBroadcastI64x4 __m512i __mm512_mask_broadcast_i64x4(__m512i s, __mmask8 k, __m256i a);
VBroadcastI64x4 __m512i __mm512_maskz_broadcast_i64x4(__mmask8 k, __m256i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions”.
EVEX-encoded instructions, syntax with reg/mem operand, see Table 2-53, “Type E6 Class Exception Conditions”.
Additionally:

#UD If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.
If EVEX.L’L’ = 0 for VPBROADCASTI32X4/VPBROADCASTI64X2.
If EVEX.L’L’ < 10b for VPBROADCASTI32X8/VPBROADCASTI64X4.
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>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>RM</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:i] := ZeroExtend(SRC[7:0])
ENDFOR
DEST[MAXVL-1:VL] := 0

**VPBROADCASTMW2D**
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j*32
    DEST[i*31:i] := ZeroExtend(SRC[15:0])
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q __m512i _mm512_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m512i _mm512_broadcastmw_epi32(__mmask16);
VPBROADCASTMB2Q __m256i _mm256_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m256i _mm256_broadcastmw_epi32(__mmask8);
VPBROADCASTMB2Q __m128i _mm_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m128i _mm_broadcastmw_epi32(__mmask8);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-54, “Type E6NF Class Exception Conditions”.
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.128.66.0F3A.W0 3F /r ib</td>
<td>A</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>VPCMPB k1 [k2], xmm2, xmm3/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 3F /r ib</td>
<td>A</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>VPCMPB k1 [k2], ymm2, ymm3/m256, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3F /r ib</td>
<td>A</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>VPCMPB k1 [k2], zmm2, zmm3/m512, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 3E /r ib</td>
<td>A</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>VPCMPUB k1 [k2], xmm2, xmm3/m128, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 3E /r ib</td>
<td>A</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>VPCMPUB k1 [k2], ymm2, ymm3/m256, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3E /r ib</td>
<td>A</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>
<tr>
<td>VPCMPUB k1 [k2], zmm2, zmm3/m512, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full Mem</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 5-8.

<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>VPPCMPNL* 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>

Table 5-8. Pseudo-Op and VPCMP* Implementation

Operation

CASE (COMPARISON PREDICATE) OF
  0: OP := EQ;
  1: OP := LT;
  2: OP := LE;
  3: OP := FALSE;
  4: OP := NEQ;
  5: OP := NLT;
  6: OP := NLE;
  7: OP := TRUE;
ESAC;

VPCMPB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k2[j] OR *no writemask*
    THEN
      CMP := SRC1[i+7:i] OP SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
      ELSE DEST[j] := 0; ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
VPCMPUB (EVEX encoded versions)

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

FOR j := 0 TO KL-1
  i := j * 8
  IF k2[j] OR *no writemask*
    THEN
      CMP := SRC1[i+7:i] OP SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
    ELSE
      DEST[j] = 0 ; zeroing-masking onlyFI;
  FI;
ENDFOR

DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPB __mmask64 _mm512_cmp_epi8_mask( __m512i a, __m512i b, int cmp);
VPCMPB __mmask64 _mm512_mask_cmp_epi8_mask( __mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPB __mmask32 _mm256_cmp_epi8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __mmask32 _mm256_mask_cmp_epi8_mask( __mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPB __mmask16 _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|ge|gt|le|lt|neq]_epi8_mask( __m512i a, __m512i b);
VPCMPB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m256i a, __m256i b);
VPCMPB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __m128i a, __m128i b);
VPCMPB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask( __mmask16 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”.
**VPCMPD/VPCMPUD—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.128.66.0F3A.W0 1F /r ib VPCMPD k1 [k2], xmm2, xmm3/m128/m32bcst, imm8</td>
<td>A</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.256.66.0F3A.W0 1F /r ib VPCMPD k1 [k2], ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</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.512.66.0F3A.W0 1F /r ib VPCMPD k1 [k2], zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A</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.128.66.0F3A.W0 1E /r ib VPCMPUD k1 [k2], xmm2, xmm3/m128/m32bcst, imm8</td>
<td>A</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.256.66.0F3A.W0 1E /r ib VPCMPUD k1 [k2], ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</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.512.66.0F3A.W0 1E /r ib VPCMPUD k1 [k2], zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A</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>Tuple Type</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>Full</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 5-8.
**Operation**

CASE (COMPARISON PREDICATE) OF
0: OP := EQ;
1: OP := LT;
2: OP := LE;
3: OP := FALSE;
4: OP := NEQ;
5: OP := NLT;
6: OP := NLE;
7: OP := TRUE;
ESAC;

**VPCMPD (EVEX encoded versions)**

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

FOR j := 0 TO KL-1
  i := j * 32
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
        ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
    ELSE DEST[j] := 0 ; zeroing-masking onlyFI;
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

**VPCMPUD (EVEX encoded versions)**

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

FOR j := 0 TO KL-1
  i := j * 32
  IF k2[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN CMP := SRC1[i+31:i] OP SRC2[31:0];
        ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];
        FI;
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
    ELSE DEST[j] := 0 ; zeroing-masking onlyFI;
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VPCMPD __mmask16 __mm512_cmp_epi32_mask( __m512i a, __m512i b, int imm);
VPCMPD __mmask16 __mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 __mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m512i a, __m512i b);
VPCMPD __mmask16 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 __mm512_cmp_epi32_mask( __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 __mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 __mm512_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m512i a, __m512i b);
VPCMPUD __mmask16 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPD __mmask8 __mm256_cmp_epi32_mask( __m256i a, __m256i b, int imm);
VPCMPD __mmask8 __mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 __mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m256i a, __m256i b);
VPCMPD __mmask8 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUD __mmask8 __mm256_cmp_epi32_mask( __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 __mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 __mm256_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m256i a, __m256i b);
VPCMPUD __mmask8 __mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 __mm_cmp_epi32_mask( __m128i a, __m128i b, int imm);
VPCMPD __mmask8 __mm_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPD __mmask8 __mm_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m128i a, __m128i b);
VPCMPD __mmask8 __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUD __mmask8 __mm_cmp_epi32_mask( __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 __mm_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 __mm_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m128i a, __m128i b);
VPCMPUD __mmask8 __mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m128i a, __m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”
VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask

**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 5-8.

---

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</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;

**VPCMPQ (EVEX encoded versions)**

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

FOR \(j := 0\) TO \(KL-1\)

\[ i := j \times 64 \]

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

\(DEST[\text{MAX\_KL-1:KL}] := 0\)

**VPCMPUQ (EVEX encoded versions)**

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

FOR \(j := 0\) TO \(KL-1\)

\[ i := j \times 64 \]

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

\(DEST[\text{MAX\_KL-1:KL}] := 0\)
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|gt|le|lt|neq]_epi64_mask(__m512i a, __m512i b);
VPCMPQ__mmask8__mm512_mask_cmp[eq|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPQ__mm512_cmp_epi64_mask(__m512i a, __m512i b, int imm);
VPCMPQ__mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ__mm512_cmp[eq|gt|le|lt|neq]_epi64_mask(__m512i a, __m512i b);
VPCMPQ__mm512_mask_cmp[eq|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions”. 
VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>OpEn</th>
<th>Tuple Type</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>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>vvvv (r)</td>
<td>ModRM:r/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 5-8.
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

VPCMPWU (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( __m256i a, __m256i b, int cmp);
VPCMPW __mmask16 __mm256_mask_cmp_epi16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask8 __mm_cmp_epi16_mask( __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 __mm512_cmp_epi16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 __mm512_mask_cmp_epi16_mask(__mmask32 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask16 __mm256_cmp_epi16_mask(__mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask16 __mm256_mask_cmp_epi16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask8 __mm_cmp_epi16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask8 __mm_mask_cmp_epi16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 __mm512_cmp_eqgegtleqltneq_epi16_mask( __m512i a, __m512i b);
VPCMPW __mmask32 __mm512_mask_cmp_eqgegtleqltneq_epi16_mask( __mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 __mm256_cmp_eqgegtleqltneq_epi16_mask( __m256i a, __m256i b);
VPCMPW __mmask16 __mm256_mask_cmp_eqgegtleqltneq_epi16_mask( __mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask8 __mm_cmp_eqgegtleqltneq_epi16_mask( __m128i a, __m128i b);
VPCMPW __mmask8 __mm_mask_cmp_eqgegtleqltneq_epi16_mask( __mmask8 m, __m128i a, __m128i b);
VPCMPUW __mmask32 __mm512_cmp_epu16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 __mm512_mask_cmp_epu16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 __mm256_cmp_epu16_mask( __m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 __mm256_mask_cmp_epu16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 __mm_cmp_epu16_mask( __m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 __mm_mask_cmp_epu16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPUW __mmask32 __mm512_cmp_eqgegtgleqltneq_epi16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 __mm512_mask_cmp_eqgegtgleqltneq_epi16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 __mm256_cmp_eqgegtgleqltneq_epi16_mask( __m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 __mm256_mask_cmp_eqgegtgleqltneq_epi16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 __mm_cmp_eqgegtgleqltneq_epi16_mask( __m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 __mm_mask_cmp_eqgegtgleqltneq_epi16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions".
**VPCOMPRESSB/VCOMPRESSW** — Store Sparse Packed Byte/Word 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 63 /r VPCOMPRESSB m128[k1], xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Compress up to 128 bits of packed byte values from xmm1 to m128 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 63 /r VPCOMPRESSB xmm1[k1][z], xmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Compress up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB m256[k1], ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Compress up to 256 bits of packed byte values from ymm1 to m256 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 63 /r VPCOMPRESSB ymm1[k1][z], ymm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Compress up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB m512[k1], zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Compress up to 512 bits of packed byte values from zmm1 to m512 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 63 /r VPCOMPRESSB zmm1[k1][z], zmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Compress up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW m128[k1], xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Compress up to 128 bits of packed word values from xmm1 to m128 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 63 /r VPCOMPRESSW xmm1[k1][z], xmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Compress up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW m256[k1], ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Compress up to 256 bits of packed word values from ymm1 to m256 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 63 /r VPCOMPRESSW ymm1[k1][z], ymm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Compress up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW m512[k1], zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Compress up to 512 bits of packed word values from zmm1 to m512 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 63 /r VPCOMPRESSW zmm1[k1][z], zmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Compress up to 512 bits of packed word values from zmm2 to zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Tuple1 Scalar</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</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 64 byte values or 32 word values from the source operand (second operand) to the destination operand (first operand), based on the active elements determined by the writemask operand. Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

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

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

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

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

**Operation**

**VPCOMPRESSB store form**

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

\( k := 0 \)

FOR \( j := 0 \) TO \( KL-1 \):

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

\[ \text{DEST}.\text{byte}[k] := \text{SRC}.\text{byte}[j] \]

\( k := k + 1 \)

**VPCOMPRESSB reg-reg form**

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

\( k := 0 \)

FOR \( j := 0 \) TO \( KL-1 \):

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

\[ \text{DEST}.\text{byte}[k] := \text{SRC}.\text{byte}[j] \]

\( k := k + 1 \)

IF *merging-masking*:

*DEST[VL-1:k*8] remains unchanged*

ELSE \( \text{DEST}[VL-1:k*8] := 0 \)

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

**VPCOMPRESSW store form**

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

\( k := 0 \)

FOR \( j := 0 \) TO \( KL-1 \):

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

\[ \text{DEST}.\text{word}[k] := \text{SRC}.\text{word}[j] \]

\( k := k + 1 \)

**VPCOMPRESSW reg-reg form**

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

\( k := 0 \)

FOR \( j := 0 \) TO \( KL-1 \):

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

\[ \text{DEST}.\text{word}[k] := \text{SRC}.\text{word}[j] \]

\( k := k + 1 \)

IF *merging-masking*:

*DEST[VL-1:k*16] remains unchanged*

ELSE \( \text{DEST}[VL-1:k*16] := 0 \)

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

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

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions".
VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register

**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 8B /r VPCOMPRESSD xmm1/m128 [k1][z], xmm2</td>
<td>A</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>A</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>A</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>Tuple Type</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>Tuple1 Scalar</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 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**

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

\[\text{SIZE} := 32\]

\[k := 0\]

\[\text{FOR } j := 0 \text{ TO } KL-1\]

\[i := j * 32\]

\[\text{IF } k1[j] \text{ OR } *\text{no controlmask}^*\]

\[\text{THEN}\]

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

\[k := k + \text{SIZE}\]

\[\text{FI}\]

\[\text{ENDFOR}\]
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

ENDFOR

IF *merging-masking*

    THEN *DEST[VL-1:k] remains unchanged*

    ELSE DEST[VL-1:k] := 0

END

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSD __m512i __mm512_mask_compress_epi32(__m512i s, __mmask16 c, __m512i a);
VPCOMPRESSD __m512i __mm512_maskz_compress_epi32(__mmask16 c, __m512i a);
VPCOMPRESSD void __mm512_mask_compressstoreu_epi32(void * a, __mmask16 c, __m512i s);
VPCOMPRESSD __m256i __m256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i __m256_maskz_compress_epi32(__mmask8 c, __m256i a);
VPCOMPRESSD void __m256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i __m128_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i __m128_maskz_compress_epi32(__mmask8 c, __m128i a);
VPCOMPRESSD void __m128_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”.

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 __m256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i __m256_maskz_compress_epi32(__mmask8 c, __m256i a);
VPCOMPRESSD void __m256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i __m128_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i __m128_maskz_compress_epi32(__mmask8 c, __m128i a);
VPCOMPRESSD void __m128_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);
VPCOMPRESSQ—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 VPCOMPRESSQ xmm1/m128 {k1}[z], xmm2</td>
<td>A</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 VPCOMPRESSQ ymm1/m256 {k1}[z], ymm2</td>
<td>A</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 VPCOMPRESSQ zmm1/m512 {k1}[z], zmm2</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</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

VPCOMPRESSQ (EVEX encoded versions) store form

(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no controlmask* THEN
    DEST[k+SIZE-1:k] := SRC[i+63:i]
    k := k + SIZE
  FI;
ENFOR
VPCOMPRESSQ (EVEX encoded versions) reg-reg form

(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE := 64
k := 0
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k] := SRC[i+63:i]
            k := k + SIZE
        FI;
ENDFOR
IF *merging-masking*
    THEN *DEST[VL-1:k] remains unchanged*
        ELSE DEST[VL-1:k] := 0
    FI
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPCOMPRESSQ __m512i _mm512_mask_compress_epi64(__m512i s, __mmask8 c, __m512i a);
VPCOMPRESSQ __m512i _mm512_maskz_compress_epi64( __mmask8 c, __m512i a);
VPCOMPRESSQ void _mm512_mask_compressstoreu_epi64(void * a, __mmask8 c, __m512i s);
VPCOMPRESSQ __m256i _mm256_mask_compress_epi64(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSQ __m256i _mm256_maskz_compress_epi64( __mmask8 c, __m256i a);
VPCOMPRESSQ void _mm256_mask_compressstoreu_epi64(void * a, __mmask8 c, __m256i s);
VPCOMPRESSQ __m128i _mm_mask_compress_epi64(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSQ __m128i _mm_maskz_compress_epi64( __mmask8 c, __m128i a);
VPCOMPRESSQ void _mm_mask_compressstoreu_epi64(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”.
# VPONFLICTD/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 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 C4 /r VPONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 C4 /r VPONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 C4 /r VPONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst</td>
<td>A V/V</td>
<td>AVX512CD</td>
<td>Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 C4 /r VPONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 C4 /r VPONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 C4 /r VPONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst</td>
<td>A V/V</td>
<td>AVX512CD</td>
<td>Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/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 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**

**VPCONFLICTD**

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

FOR j := 0 TO KL-1
  i := j*32
  IF MaskBit(j) OR *no writemask* THEN
    FOR k := 0 TO j-1
      m := k*32
      IF ((SRC[i+31:i] = SRC[m+31:m])) THEN
        DEST[i+k] := 1
      ELSE
        DEST[i+k] := 0
      FI
    ENDFOR
    DEST[i+31:i+j] := 0
  ELSE
    IF *merging-masking* THEN
      *DEST[i+31:i] remains unchanged*
    ELSE
      DEST[i+31:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

**VPCONFLICTQ**

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

FOR j := 0 TO KL-1
  i := j*64
  IF MaskBit(j) OR *no writemask* THEN
    FOR k := 0 TO j-1
      m := k*64
      IF ((SRC[i+63:i] = SRC[m+63:m])) THEN
        DEST[i+k] := 1
      ELSE
        DEST[i+k] := 0
      FI
    ENDFOR
    DEST[i+63:i+j] := 0
  ELSE
    IF *merging-masking* THEN
      *DEST[i+63:i] remains unchanged*
    ELSE
      DEST[i+63:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPCONFICTD __m512i _mm512_conflict_epi32( __m512i a);
VPCONFICTD __m512i _mm512_mask_conflict_epi32( __m512i s, __mmask16 m, __m512i a);
VPCONFICTD __m512i _mm512_maskz_conflict_epi32( __mmask16 m, __m512i a);
VPCONFICTQ __m512i _mm512_conflict_epi64( __m512i a);
VPCONFICTQ __m512i _mm512_mask_conflict_epi64( __m512i s, __mmask8 m, __m512i a);
VPCONFICTQ __m512i _mm512_maskz_conflict_epi64( __mmask8 m, __m512i a);
VPCONFICTD __m256i _mm256_conflict_epi32( __m256i a);
VPCONFICTD __m256i _mm256_mask_conflict_epi32( __m256i s, __mmask8 m, __m256i a);
VPCONFICTD __m256i _mm256_maskz_conflict_epi32( __mmask8 m, __m256i a);
VPCONFICTQ __m256i _mm256_conflict_epi64( __m256i a);
VPCONFICTQ __m256i _mm256_mask_conflict_epi64( __m256i s, __mmask8 m, __m256i a);
VPCONFICTQ __m256i _mm256_maskz_conflict_epi64( __mmask8 m, __m256i a);
VPCONFICTD __m128i _mm_conflict_epi32( __m128i a);
VPCONFICTD __m128i _mm_mask_conflict_epi32( __m128i s, __mmask8 m, __m128i a);
VPCONFICTD __m128i _mm_maskz_conflict_epi32( __mmask8 m, __m128i a);
VPCONFICTQ __m128i _mm_conflict_epi64( __m128i a);
VPCONFICTQ __m128i _mm_mask_conflict_epi64( __m128i s, __mmask8 m, __m128i a);
VPCONFICTQ __m128i _mm_maskz_conflict_epi64( __mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions”.

VPCONFICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/ Register
VPDPBUSD — Multiply and Add Unsigned and Signed Bytes

<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 50 /r VPDPBUSD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1[k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 4 pairs of signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 4 pairs of signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 50 /r VPDPBUSD zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI</td>
<td>Multiply groups of 4 pairs of signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result in zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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**

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

This instruction supports memory fault suppression.
**Operation**

**VPDPBUSD dest, src1, src2 (VEX encoded versions)**

VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:

    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
    DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

DEST[MAX_VL-1:VL] := 0

**VPDPBUSD dest, src1, src2 (EVEX encoded versions)**

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

ORIGDEST := DEST
FOR i := 0 TO KL-1:

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

DEST[MAX_VL-1:VL] := 0

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

VPDPBUSD __m128i _mm_dpbusd_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_dpbusd_epi32(__m128i, __m128i, __m128i);
VPDPBUSD __m128i _mm_mask_dpbusd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPBUSD __m128i _mm_maskz_dpbusd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPBUSD __m256i _mm256_dpbusd_avx_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_dpbusd_epi32(__m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_mask_dpbusd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_maskz_dpbusd_epi32(__mmask16, __m256i, __m256i, __m256i);
VPDPBUSD __m512i _mm512_dpbusd_epi32(__m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_mask_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_maskz_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);
SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions”.
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions”.
## VPDPBUSDS — Multiply and Add Unsigned and Signed Bytes with Saturation

<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 51 /r VPDPBUSDS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 4 pairs signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 4 pairs signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 51 /r VPDPBUSDS xmm1[k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI</td>
<td>Multiply groups of 4 pairs signed bytes in xmm3/m128/m32bcst with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI</td>
<td>Multiply groups of 4 pairs signed bytes in ymm3/m256/m32bcst with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 51 /r VPDPBUSDS zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI</td>
<td>Multiply groups of 4 pairs signed bytes in zmm3/m512/m32bcst with corresponding unsigned bytes of zmm2, summing those products and adding them to doubleword result, with signed saturation in zmm1, under writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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

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

This instruction supports memory fault suppression.
Operation

**VPDPBUSDS dest, src1, src2 (VEX encoded versions)**

VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:

// Extending to 16b
// src1extend := ZERO_EXTEND
// src2extend := SIGN_EXTEND

p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)

DEST[MAX_VL-1:VL] := 0

**VPDPBUSDS dest, src1, src2 (EVEX encoded versions)**

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

ORIGDEST := DEST
FOR i := 0 TO KL-1:

IF k1[i] or *no writemask*:

// Byte elements of SRC1 are zero-extended to 16b and
// byte elements of SRC2 are sign extended to 16b before multiplication.

IF SRC2 is memory and EVEX.b == 1:
    t := SRC2.dword[0]
ELSE:
    t := SRC2.dword[i]
p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) * SIGN_EXTEND(t.byte[3])
DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
ELSE IF *zeroing*:
    DEST.dword[i] := 0
ELSE: // Merge masking, dest element unchanged
    DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPDPBUSDS _m128i _mm_dpbusds_avx_epi32(__m128i __m128i __m128i);
VPDPBUSDS _m128i _mm_dpbusds_epi32(__m128i __m128i __m128i);
VPDPBUSDS _m128i _mm_mask_dpbusds_epi32(__m128i __m128i __m128i);
VPDPBUSDS _m128i _mm_maskz_dpbusds_epi32(__mmmask8 __m128i __m128i);
VPDPBUSDS __m256i __mm256_dpbusds_avx_epi32(__m256i __m256i __m256i);
VPDPBUSDS __m256i __mm256_dpbusds_epi32(__m256i __m256i __m256i);
VPDPBUSDS __m256i __mm256_mask_dpbusds_epi32(__m256i __m256i __m256i);
VPDPBUSDS __m256i __mm256_maskz_dpbusds_epi32(__mmmask8 __m256i __m256i);
VPDPBUSDS __m512i __mm512_dpbusds_epi32(__m512i __m512i __m512i);
VPDPBUSDS __m512i __mm512_mask_dpbusds_epi32(__m512i __m512i __m512i);
VPDPBUSDS __m512i __mm512_maskz_dpbusds_epi32(__mmmask16 __m512i __m512i);
SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions".
EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions".
## VPDPWSSD — Multiply and Add Signed Word Integers

### Instruction operand encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>NA</td>
<td>ModRM:reg(r, w)</td>
<td>VEX.vvvv(r)</td>
<td>ModRM:r/m(r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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

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

This instruction supports memory fault suppression.

### Operation

\[
\text{VPDPWSSD dest, src1, src2 (VEX encoded versions)}
\]

\[
\text{VL}=(128, 256) \\
\text{KL} = \text{VL}/32 \\
\text{ORIGDEST} := \text{DEST} \\
\text{FOR} i := 0 \text{ TO } \text{KL}-1: \\
\begin{align*}
  p1\text{dword} & := \text{SIGN\_EXTEND}(\text{SRC1}.\text{word}[2*i+0]) \times \text{SIGN\_EXTEND}(\text{SRC2}.\text{word}[2*i+0]) \\
  p2\text{dword} & := \text{SIGN\_EXTEND}(\text{SRC1}.\text{word}[2*i+1]) \times \text{SIGN\_EXTEND}(\text{SRC2}.\text{word}[2*i+1]) \\
  \text{DEST}.\text{dword}[i] & := \text{ORIGDEST}.\text{dword}[i] + p1\text{dword} + p2\text{dword} \\
  \text{DEST}[\text{MAX\_VL}-1:VL] & := 0
\end{align*}
\]
VPDPWSSD dest, src1, src2 (EVEX encoded versions)
(KL,VL)=(4,128), (8,256), (16,512)

ORIGDEST := DEST
FOR i := 0 TO KL-1:
  IF k[i] or "no writemask": 
    IF SRC2 is memory and EVEX.b == 1:
      t := SRC2.dword[0]
    ELSE:
      t := SRC2.dword[i]
  p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
  p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
  DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
ELSE IF "zeroing":
  DEST.dword[i] := 0
ELSE: /* Merge masking, dest element unchanged */
  DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPDPWSSD __m128i _mm_dpwssd_avx_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_dpwssd_epi32(__m128i, __m128i, __m128i);
VPDPWSSD __m128i _mm_mask_dpwssd_epi32(__m128i, __mmask8, __m128i, __m128i);
VPDPWSSD __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPWSSD __m256i _mm256_dpwssd_avx_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_dpwssd_epi32(__m256i, __m256i, __m256i);
VPDPWSSD __m256i _mm256_mask_dpwssd_epi32(__m256i, __mmask8, __m256i, __m256i);
VPDPWSSD __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPWSSD __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i);
VPDPWSSD __m512i _mm512_mask_dpwssd_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPWSSD __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions".
EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions".
# VPDPWSSDS — Multiply and Add Signed Word Integers with Saturation

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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

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

This instruction supports memory fault suppression.
Operation

VPDPwSSDS dest, src1, src2 (VEX encoded versions)
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    p1dword := SIGN_EXTEND(SRC1.word[2*i+0]) * SIGN_EXTEND(SRC2.word[2*i+0])
    p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(SRC2.word[2*i+1])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
DEST[MAX_VL-1:VL] := 0

VPDPwSSDS dest, src1, src2 (EVEX encoded versions)
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    IF k1[i] or *no writemask*:
        IF SRC2 is memory and EVEX.b == 1:
            t := SRC2.dword[0]
        ELSE:
            t := SRC2.dword[i]
        p1dword := SIGN_EXTEND(SRC1.word[2*i]) * SIGN_EXTEND(t.word[0])
        p2dword := SIGN_EXTEND(SRC1.word[2*i+1]) * SIGN_EXTEND(t.word[1])
        DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
    ELSE IF *zeroing*:
        DEST.dword[i] := 0
    ELSE: // Merge masking, dest element unchanged
        DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPDPwSSDS __m128i _mm_dpwssds_avx_epi32(__m128i, __m128i, __m128i);
VPDPwSSDS __m128i _mm_dpwssds_epi32(__m128i, __m128i, __m128i);
VPDPwSSDS __m128i _mm_mask_dpwssds_epi32(__m128i, __m128i, __m128i, __mmask8);
VPDPwSSDS __m128i _mm_maskz_dpwssd_epi32(__mmask8, __m128i, __m128i, __m128i);
VPDPwSSDS __m256i _mm256_dpwssds_avx_epi32(__m256i, __m256i, __m256i);
VPDPwSSDS __m256i _mm256_dpwssds_epi32(__m256i, __m256i, __m256i);
VPDPwSSDS __m256i _mm256_mask_dpwssd_epi32(__m256i, __m256i, __m256i, __mmask8);
VPDPwSSDS __m256i _mm256_maskz_dpwssd_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPwSSDS __m512i _mm512_dpwssd_epi32(__m512i, __m512i, __m512i, __m512i);
VPDPwSSDS __m512i _mm512_mask_dpwssd_epi32(__m512i, __m512i, __mmask16, __m512i, __m512i);
VPDPwSSDS __m512i _mm512_maskz_dpwssd_epi32(__mmask16, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions
None.

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions".
EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions".
VPERM2F128 — Permute Floating-Point Values

**Opcode/ Instruction** | Op/ En | 64/32 bit Mode Support | CPUID Feature Flag | Description  
--- | --- | --- | --- | ---  
VEX.256.66.0F3A.W0 06 /r ib | RVMI | V/V | AVX | Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.  
VPERM2F128 ymm1, ymm2, ymm3/m256, imm8 |  |  |  |  

**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 (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8</td>
</tr>
</tbody>
</table>

**Description**

Permutes 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](image)

**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]
2: DEST[127:0] := SRC2[127:0]
ESAC
CASE IMM8[5:4] of
0: DEST[255:128] := SRC1[127:0]
ESAC
IF (imm8[3])
DEST[127:0] := 0
FI
IF (imm8[7])
DEST[MAXVL-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 Table 2-23, "Type 6 Class Exception Conditions"; 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.256.66.0F3A.W0 46 /r ib VPERM2I128 ymm1, ymm2, ymm3/m256, imm8</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>
</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 (r)</td>
<td>ModRM:r/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]
2: DEST[127:0] := SRC2[127:0]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] := SRC1[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 Table 2-23, "Type 6 Class Exception Conditions"; additionally:

- #UD If VEX.L = 0,
  - If VEX.W = 1.
### VPERMB—Permute Packed Bytes 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>EVEX.128.66.0F38.W0 8D /r VPERMB xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in xmm3/m128 using byte indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8D /r VPERMB ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in ymm3/m256 using byte indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8D /r VPERMB zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>A V/V</td>
<td>AVX512_VBMI</td>
<td>Permute bytes in zmm3/m512 using byte indexes in zmm2 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>Tuple Type</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>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM1/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

#### Description

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

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

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

#### Operation

**VPERMB (EVEX encoded versions)**

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

If VL = 128:

- n := 3;

Else if VL = 256:

- n := 4;

Else if VL = 512:

- n := 5;

Fi;

For j := 0 To KL-1:

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

If k1[j] OR *no writemask* THEN

- DEST[*8 + 7:j*8] := SRC2[id*8 +7: id*8];

Else if zeroing-masking THEN

- DEST[*8 + 7:j*8] := 0;

*ELSE

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

Fi

EndFor

DEST[MAX_VL-1:VL] := 0;

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

VPERMB __m512i __mm512_permutexvar_epi8(__m512i idx, __m512i a);
VPERMB __m512i _mm512_mask_permutexvar_epi8(__m512i s, __mmask64 k, __m512i idx, __m512i a);
VPERMB __m512i _mm512_maskz_permutexvar_epi8( __mmask64 k, __m512i idx, __m512i a);
VPERMB __m256i _mm256_permutexvar_epi8( __m256i idx, __m256i a);
VPERMB __m256i _mm256_mask_permutexvar_epi8( __m256i s, __mmask32 k, __m256i idx, __m256i a);
VPERMB __m256i _mm256_maskz_permutexvar_epi8( __mmask32 k, __m256i idx, __m256i a);
VPERMB __m128i _mm_permutexvar_epi8( __m128i idx, __m128i a);
VPERMB __m128i _mm_mask_permutexvar_epi8( __m128i s, __mmask16 k, __m128i idx, __m128i a);
VPERMB __m128i _mm_maskz_permutexvar_epi8( __mmask16 k, __m128i idx, __m128i a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions”.
### 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.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.W0 36 /r VPERMD ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</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.512.66.0F38.W0 36 /r VPERMD zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</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.128.66.0F38.W1 8D /r VPERMW xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>C</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.256.66.0F38.W1 8D /r VPERMW ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>C</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.512.66.0F38.W1 8D /r VPERMW zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>C</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>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full Mem</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 doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

**VEX.256 encoded VPERMD**: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

**EVEX encoded VPERMD**: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

**VPERMW**: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

**EVEX.128 encoded versions**: Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.
### Operation

**VPERMD (EVEX encoded versions)**

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

IF VL = 256 THEN n := 2; FI;
IF VL = 512 THEN n := 3; FI;

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

DEST[MAXVL-1:VL] := 0

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

DEST[31:0] := (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] := (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] := (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] := (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] := (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[255:224] := (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] := 0

**VPERMW (EVEX encoded versions)**

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

IF VL = 128 THEN n := 2; FI;
IF VL = 256 THEN n := 3; FI;
IF VL = 512 THEN n := 4; FI;

FOR j := 0 TO KL-1
i := j * 16
id := 16*SRC1[i+n:i]
IF k1[j] OR “no writemask”
THEN DEST[i+15:i] := SRC2[id+15:id]
ELSE
   IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+15:i] := 0
      FI
   FI;
ENDFOR

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPERMD __m512i __m512_permutexvar_epi32( __m512i idx, __m512i a);
VPERMD __m512i __m512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i __m512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i __m256_permutexvar_epi32( __m256i idx, __m256i a);
VPERMD __m256i __m256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i __m256_maskz_permutexvar_epi32( __mmask8 k, __m256i idx, __m256i a);
VPERMD __m128i __m128_permutexvar_epi32( __m128i idx, __m128i a);
VPERMD __m128i __m128_mask_permutexvar_epi32(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMD __m128i __m128_maskz_permutexvar_epi32( __mmask8 k, __m128i idx, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions".
EVEX-encoded VPERMD, see Table 2-50, "Type E4NF Class Exception Conditions".
EVEX-encoded VPERMW, see Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions".
Additionally:

#UD If VEX.L = 0.
If EVEX.L'L = 0 for VPERMD.
VPERM12B—Full Permute of Bytes 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.128.66.0F38.W0 75 /r VPERM12B xmm1 [k1][z], xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in xmm3/m128 and xmm2 using byte indexes in xmm1 and store the byte results in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 75 /r VPERM12B ymm1 [k1][z], ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in ymm3/m256 and ymm2 using byte indexes in ymm1 and store the byte results in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 75 /r VPERM12B zmm1 [k1][z], zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI</td>
<td>Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full Mem</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**

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

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

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

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.
Operation

VPERMIB (EVEX encoded versions)

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

IF VL = 128:
  id := 3;
ELSE IF VL = 256:
  id := 4;
ELSE IF VL = 512:
  id := 5;
FI;

TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
  off := 8*SRC1[j*8 + id: j*8] ;
  IF k1[j] OR *no writemask*:
    DEST[j*8 + 7: j*8] := TMP_DEST[j*8+id+1]? SRC2[off+7:off] : SRC1[off+7:off];
  ELSE IF *zeroing-masking*
    DEST[j*8 + 7: j*8] := 0;
  *ELSE
    DEST[j*8 + 7: j*8] remains unchanged*
  FI;
ENDFOR

DEST[MAX_VL-1:VL] := 0;

Intel C/C++ Compiler Intrinsic Equivalent

VPERMIB __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMIB __m512i _mm512_mask2_permutex2var_epi8(__m512i a, __m512i idx, __mmask64 k, __m512i b);
VPERMIB __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMIB __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMIB __m256i _mm256_mask2_permutex2var_epi8(__m256i a, __m256i idx, __mmask32 k, __m256i b);
VPERMIB __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMIB __m128i _mm128i_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMIB __m128i _mm128i_mask2_permutex2var_epi8(__m128i a, __m128i idx, __mmask16 k, __m128i b);
VPERMIB __m128i _mm128i_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions".
<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 75 /r VPERM1W xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A / 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 75 /r VPERM1W ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A / 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 75 /r VPERM1W zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A / 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 76 /r VPERM1D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B / 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 76 /r VPERM1D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B / 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 76 /r VPERM1D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B / 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 76 /r VPERM1Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B / 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 76 /r VPERM1Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B / 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 76 /r VPERM1Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B / 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 77 /r VPERMI2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B / 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 77 /r VPERMI2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B / 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 77 /r VPERMI2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B / 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>
<td></td>
</tr>
</tbody>
</table>
Permutes 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

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

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

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

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.
Operation

**VPERM2W (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
      ELSE
          IF *merging-masking* ; merging-masking
             THEN *DEST[i+15:i] remains unchanged*
          ELSE ; zeroing-masking
             DEST[i+15:i] := 0
          FI
      FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VPERM2D/VPERM2PS (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
            ELSE
         FI
      ELSE
      FI;
ENDFOR
ELSE
 IF *merging-masking*  ; merging-masking
 THEN *DEST[i+31:i] remains unchanged*
 ELSE  ; zeroing-masking
 DEST[i+31:i] := 0
 FI
 FI
ENDFOR
DEST[MAXVL-1:VL] := 0

**VPERMI2Q/VPERMI2PD (EVEX encoded versions)**

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

IF VL = 128
 id := 0
 FI
IF VL = 256
 id := 1
 FI
IF VL = 512
 id := 2
 FI
TMP_DEST:= DEST
FOR j := 0 TO KL-1
 i := j * 64
 off := 64*TMP_DEST[i+id:i]
 IF k1[j] OR *no writemask*
 THEN
 IF (EVEX.b = 1) AND (SRC2 *is memory*)
 THEN
 : SRC1[off+63:off]
 ELSE
 : SRC1[off+63:off]
 FI
 ELSE
 IF *merging-masking*  ; merging-masking
 THEN *DEST[i+63:i] remains unchanged*
 ELSE  ; zeroing-masking
 DEST[i+63:i] := 0
 FI
 ENDIF
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMI2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMI __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMI2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMI2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMI2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMI2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMI2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMI2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMI2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMI2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERMI2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMI2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMI2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMI2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
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, __m512i idx, __mmask32 k, __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 Table 2-50, “Type E4NF Class Exception Conditions”.
VPERM2W: See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions”.
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.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 0D /r VPERMILPD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 0D /r VPERMILPD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>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>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 0D /r VPERMILPD zmm1 (k1){z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>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>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute double-precision floating-point values in xmm2/m128 using controls from imm8.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 05 /r ib VPERMILPD ymm1, ymm2/m256, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute double-precision floating-point values in ymm2/m256 using controls from imm8.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 05 /r ib VPERMILPD xmm1 (k1){z}, xmm2/m128/m64bcst, imm8</td>
<td>D</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 05 /r ib VPERMILPD ymm1 (k1){z}, ymm2/m256/m64bcst, imm8</td>
<td>D</td>
<td>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>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 05 /r ib VPERMILPD zmm1 (k1){z}, zmm2/m512/m64bcst, imm8</td>
<td>D</td>
<td>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>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full</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 (MAXVL-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 \((\text{SRC1 is memory})\)

THEN \(\text{TMP}\_\text{SRC1}[i+63:i] := \text{SRC1}[63:0]\);

ELSE \(\text{TMP}\_\text{SRC1}[i+63:i] := \text{SRC1}[i+63:i]\);

FI;

ENDFOR;

IF \((\text{imm8}[0] = 0)\) THEN \(\text{TMP}\_\text{DEST}[63:0] := \text{SRC1}[63:0];\) FI;

IF \((\text{imm8}[0] = 1)\) THEN \(\text{TMP}\_\text{DEST}[63:0] := \text{TMP}\_\text{SRC1}[127:64];\) FI;

IF \((\text{imm8}[1] = 0)\) THEN \(\text{TMP}\_\text{DEST}[127:64] := \text{TMP}\_\text{SRC1}[63:0];\) FI;

IF \((\text{imm8}[1] = 1)\) THEN \(\text{TMP}\_\text{DEST}[127:64] := \text{TMP}\_\text{SRC1}[127:64];\) FI;

IF \((\text{imm8}[2] = 0)\) THEN \(\text{TMP}\_\text{DEST}[191:128] := \text{TMP}\_\text{SRC1}[191:128];\) FI;

IF \((\text{imm8}[2] = 1)\) THEN \(\text{TMP}\_\text{DEST}[191:128] := \text{TMP}\_\text{SRC1}[255:192];\) FI;

IF \((\text{imm8}[3] = 0)\) THEN \(\text{TMP}\_\text{DEST}[255:192] := \text{TMP}\_\text{SRC1}[191:128];\) FI;

IF \((\text{imm8}[3] = 1)\) THEN \(\text{TMP}\_\text{DEST}[255:192] := \text{TMP}\_\text{SRC1}[255:192];\) FI;

IF \((\text{imm8}[4] = 0)\) THEN \(\text{TMP}\_\text{DEST}[319:256] := \text{TMP}\_\text{SRC1}[319:256];\) FI;

IF \((\text{imm8}[4] = 1)\) THEN \(\text{TMP}\_\text{DEST}[319:256] := \text{TMP}\_\text{SRC1}[383:320];\) FI;

IF \((\text{imm8}[5] = 0)\) THEN \(\text{TMP}\_\text{DEST}[383:320] := \text{TMP}\_\text{SRC1}[319:256];\) FI;

IF \((\text{imm8}[5] = 1)\) THEN \(\text{TMP}\_\text{DEST}[383:320] := \text{TMP}\_\text{SRC1}[383:320];\) FI;

IF \((\text{imm8}[6] = 0)\) THEN \(\text{TMP}\_\text{DEST}[447:384] := \text{TMP}\_\text{SRC1}[447:384];\) FI;

IF \((\text{imm8}[6] = 1)\) THEN \(\text{TMP}\_\text{DEST}[447:384] := \text{TMP}\_\text{SRC1}[511:448];\) FI;

IF \((\text{imm8}[7] = 0)\) THEN \(\text{TMP}\_\text{DEST}[511:448] := \text{TMP}\_\text{SRC1}[447:384];\) FI;

IF \((\text{imm8}[7] = 1)\) THEN \(\text{TMP}\_\text{DEST}[511:448] := \text{TMP}\_\text{SRC1}[511:448];\) FI;

FI;

FOR \(j := 0 \) TO \(KL-1\)

\[i := j * 64\]

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

THEN \(\text{DEST}[i+63:i] := \text{TMP}\_\text{DEST}[i+63:i]\)

ELSE

IF \("merging-masking"\)

THEN \("\text{DEST}[i+63:i] remains unchanged"\)

ELSE \("zeroing-masking"

\(\text{DEST}[i+63:i] := 0\)

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)

VPERMILPD (256-bit immediate version)

\(\text{IF} \ (\text{imm8}[0] = 0) \ \text{THEN} \ \text{DEST}[63:0] := \text{SRC1}[63:0]\)

\(\text{IF} \ (\text{imm8}[0] = 1) \ \text{THEN} \ \text{DEST}[63:0] := \text{SRC1}[127:64]\)

\(\text{IF} \ (\text{imm8}[1] = 0) \ \text{THEN} \ \text{DEST}[127:64] := \text{SRC1}[63:0]\)

\(\text{IF} \ (\text{imm8}[1] = 1) \ \text{THEN} \ \text{DEST}[127:64] := \text{SRC1}[127:64]\)

\(\text{IF} \ (\text{imm8}[2] = 0) \ \text{THEN} \ \text{DEST}[191:128] := \text{SRC1}[191:128]\)

\(\text{IF} \ (\text{imm8}[2] = 1) \ \text{THEN} \ \text{DEST}[191:128] := \text{SRC1}[255:192]\)

\(\text{IF} \ (\text{imm8}[3] = 0) \ \text{THEN} \ \text{DEST}[255:192] := \text{SRC1}[191:128]\)

\(\text{IF} \ (\text{imm8}[3] = 1) \ \text{THEN} \ \text{DEST}[255:192] := \text{SRC1}[255:192]\)

\(\text{DEST}[\text{MAXVL}-1:256] := 0\)
VPERMILPD (128-bit immediate version)
IF (imm8[0] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] := SRC1[63:0]
DEST[MAXVL-1:128] := 0

VPERMILPD (EVEX variable versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] := SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
    FI;
ENDFOR;
IF (TMP_SRC2[1] = 0) THEN TMP_DEST[63:0] := SRC1[63:0]; FI;
IF (TMP_SRC2[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 (EVEX.b = 1) AND (SRC2 *is memory*)
  FI;
IF (EVEX.b = 1) AND (SRC2 *is memory*)
  THEN TMP_DEST[319:256] := SRC1[319:256];
  ELSE TMP_DEST[319:256] := SRC1[383:320];
  FI;
IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] := SRC1[319:256]; FI;
IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] := SRC1[447:384]; FI;
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
      FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
VPERMILPD (256-bit variable version)
IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:256] := 0

VPERMILPD (128-bit variable version)
IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPERMILPD __m512d _mm512_permute_pd(__m512d a, int imm);
VPERMILPD __m512d _mm512_mask_permute_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d _mm512_maskz_permute_pd(__mmask8 k, __m512d a, int imm);
VPERMILPD __m256d _mm256_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);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions”; additionally:
#UD If VEX.W = 1.
EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions”; additionally:
#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.128.66.0F38.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>VEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1, xmm2/m128, imm8</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1, ymm2/m256, imm8</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 0C /r VPERMILPS xmm1 (k1){z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 0C /r VPERMILPS ymm1 (k1){z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</td>
<td>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>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 0C /r VPERMILPS zmm1 (k1){z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1 (k1){z}, xmm2/m128/m32bcst, imm8</td>
<td>D</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1 (k1){z}, ymm2/m256/m32bcst, imm8</td>
<td>D</td>
<td>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>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 04 /r ib VPERMILPS zmm1 (k1){z}, zmm2/m512/m32bcst, imm8</td>
<td>D</td>
<td>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>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>D</td>
<td>Full</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]);

IF VL >= 256
    FI;

IF VL >= 512
    TMP_DEST[287:256] := Select4(TMP_SRC1[383:256], imm8[1:0]);
    TMP_DEST[415:384] := Select4(TMP_SRC1[511:384], imm8[1:0]);
    TMP_DEST[511:480] := Select4(TMP_SRC1[511:384], imm8[7:6]);
    FI;

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

DEST[MAXVL-1:VL] := 0
VPERMILPS (256-bit immediate version)
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[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]);

VPERMILPS (128-bit immediate version)
DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[MAXVL-1:128] := 0

VPERMILPS (EVEX variable versions)

(KL, VL) = (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] := SRC2[31:0];
        ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
    FI;
ENDFOR;
TMP_DEST[31:0] := Select4(SRC1[127:0], TMP_SRC2[1:0]);
TMP_DEST[63:32] := Select4(SRC1[127:0], TMP_SRC2[33:32]);
TMP_DEST[95:64] := Select4(SRC1[127:0], TMP_SRC2[65:64]);
TMP_DEST[127:96] := Select4(SRC1[127:0], TMP_SRC2[97:96]);
IF VL >= 256
FI;
IF VL >= 512
    TMP_DEST[479:448] := Select4(SRC1[511:384], TMP_SRC2[449:448]);
    TMP_DEST[511:480] := Select4(SRC1[511:384], TMP_SRC2[481:480]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] := 0 ;zeroing-masking
            FI;
    FI;
ENDFOR;
VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

Fl;

Fl;
ENDFOR
DESTM[\text{MAXVL-1:VL}] := 0

VPERMILPS (256-bit variable version)
DESTM[31:0] := Select4(SRCC[127:0], SRC[1:0]);
DESTM[63:32] := Select4(SRCC[127:0], SRC[33:32]);
DESTM[95:64] := Select4(SRCC[127:0], SRC[65:64]);
DESTM[127:96] := Select4(SRCC[127:0], SRC[97:96]);
DESTM[\text{MAXVL-1:256}] := 0

VPERMILPS (128-bit variable version)
DESTM[31:0] := Select4(SRCC[127:0], SRC[1:0]);
DESTM[63:32] := Select4(SRCC[127:0], SRC[33:32]);
DESTM[95:64] := Select4(SRCC[127:0], SRC[65:64]);
DESTM[127:96] := Select4(SRCC[127:0], SRC[97:96]);
DESTM[\text{MAXVL-1:128}] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPS __m512 _mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 _mm512_mask_permute_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 _mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 _mm256_permute_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 _mm256_mask_permute_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 _mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 _mm128_permute_ps(___m128 a, int imm);
VPERMILPS __m128 _mm128_maskz_permute_ps( ___mmask8 k, ___m128 a, int imm);
VPERMILPS __m512 _mm512_permutevar_ps(__m512i i, __m512 a);
VPERMILPS __m512 _mm512_mask_permutevar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m512 _mm512_maskz_permutevar_ps( __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m256 _mm256_permutevar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 _mm256_mask_permutevar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 _mm256_maskz_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 _mm128_permutevar_ps(___m128 a, ___m128i control);
VPERMILPS __m128 _mm128_maskz_permutevar_ps( ___mmask8 k, ___m128 i, ___m128 a);
VPERMILPS __m128 _mm128_permute_ps(___m128 a, int control);
VPERMILPS __m256 _mm256_permute_ps(___m256 a, int control);
VPERMILPS __m128 _mm128_permutevar_ps(___m128 a, ___m128i control);
VPERMILPS __m256 _mm256_permutevar_ps(___m256 a, ___m256i control);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions”; additionally:
#UD If VEX.W = 1.
EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions”; additionally:
#UD If either (E)VEX.vvvv != 1111B and with imm8.
VPERMPD—Permute Double-Precision Floating-Point Elements

### Opcode/ Instruction

**VEX.256.66.0F3A.W1 01 /r ib**  
VPERMPD ymm1, ymm2/m256, imm8  
**Op / En** A  
**64/32 bit Mode Support** V/V  
**CPUID Feature Flag** AVX2  
**Description** Permute double-precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.

**EVEX.512.66.0F3A.W1 01 /r ib**  
VPERMPD ymm1 {k1}{z}, zmm2/m512/m64bcst, imm8  
**Op / En** B  
**64/32 bit Mode Support** V/V  
**CPUID Feature Flag** AVX512F  
**Description** Permute double-precision floating-point elements in zmm2/m512/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1.

**EVEX.256.66.0F38.W1 16 /r**  
VPERMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst  
**Op / En** C  
**64/32 bit Mode Support** V/V  
**CPUID Feature Flag** AVX512F  
**Description** Permute double-precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMr/m (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</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 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 quadword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a quadword 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
      FI;
  FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VPERMPD (EVEX - vector control forms)**

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

FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] := SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
  FI;

ENDFOR;

IF VL = 256
  TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 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[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
TMP_DEST[319:256] := (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
TMP_DEST[447:384] := (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
TMP_DEST[511:448] := (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];

Fi;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN DEST[i+63:i] := TMP_DEST[i+63:i]
  ELSE IF *merging-masking* THEN *DEST[i+63;i] remains unchanged*
  ELSE DEST[i+63:i] := 0 ;zeroing-masking
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMPD (VEX.256 encoded version)
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPERMPD __m512d _mm512_permutex_pd(__m512d a, int imm);
VPERMPD __m512d __m512d_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d __m512d_maskz_permutex_pd(__mmask16 k, __m512d a, int imm);
VPERMPD __m512d __m512d_permutexvar_pd(__m512d s, __mmask16 k, __m512d i, __m512d a);
VPERMPD __m512d __m512d_permutexvar_pd(__m512d s, __mmask16 k, __m512d i, __m512d a);
VPERMPD __m256d __m256d_permutex_epi64(__m256d a, int imm);
VPERMPD __m256d __m256d_permutex_epi64(__m256d s, __mmask8 k, __m256d a, int imm);
VPERMPD __m256d __m256d_permutexvar_epi64(__m256d s, __mmask8 k, __m256d i, __m256d a);
VPERMPD __m256d __m256d_permutexvar_epi64(__m256d s, __mmask8 k, __m256d i, __m256d a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions"; additionally:
#UD IF VEX.L = 0.
   IF VEX.vvvv != 1111B.
EVEX-encoded instruction, see Table 2-50, "Type E4NF Class Exception Conditions"; additionally:
#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>A</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.256.66.0F38.W0 16 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision floating-point elements in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 subject to write mask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 16 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvv v (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (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 (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

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

Operation

**VPERMPS (EVEX forms)**

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

FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] := SRC2[31:0];
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i];
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[31:0] := (TMP_SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
  TMP_DEST[63:32] := (TMP_SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
  TMP_DEST[95:64] := (TMP_SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
  TMP_DEST[127:96] := (TMP_SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
VPERMPS—Permute Single-Precision Floating-Point Elements

### INSTRUCTION SET REFERENCE, V-Z

**Vol. 2C 5-385**

**FI:**

**IF** `VL = 512`

```plaintext
TMP_DEST[31:0] := (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
TMP_DEST[95:64] := (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
TMP_DEST[127:96] := (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
TMP_DEST[287:256] := (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
TMP_DEST[447:416] := (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
TMP_DEST[479:448] := (TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
TMP_DEST[511:480] := (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];
```

**FI:**

**FOR** `j := 0 TO KL-1`

```plaintext
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[MAXVL-1:VL] := 0**

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

```plaintext
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[255:224] := (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] := 0
```

---

**VPERMPS—Permute Single-Precision Floating-Point Elements**

Vol. 2C 5-385
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(__m256 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 Table 2-21, “Type 4 Class Exception Conditions”; additionally:
#UD If VEX.L = 0.

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions”.
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 VPERMQ ymm1, ymm2/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Permutes qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 00 / r ib VPERMQ ymm1 [k1][z], ymm2/m256/m64bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permutes qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 00 / r ib VPERMQ zmm1 [k1][z], zmm2/m512/m64bcst, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permutes qwords in zmm2/m512/m64bcst using indexes in imm8 and store the result in zmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 36 / r VPERMQ ymm1 [k1][z], ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permutes qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 36 / r VPERMQ zmm1 [k1][z], zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permutes qwords in zmm3/m512/m64bcst using indexes in zmm2 and store the result in zmm1.</td>
</tr>
</tbody>
</table>

**InstructionOperand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMreg/r (r)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg/r (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 (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

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

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.
**Operation**

**VPERMQ (EVEX - imm8 control forms)**

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

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

TMP_DEST[63:0] := (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] := (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];

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[MAXVL-1:VL] := 0

**VPERMQ (EVEX - vector control forms)**

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

FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] := SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
  TMP_DEST[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[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
FI;
\[
\begin{align*}
\text{TMP\_DEST}[319:256] & := (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[258:256] \times 64))[63:0]; \\
\text{TMP\_DEST}[383:320] & := (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[322:320] \times 64))[63:0]; \\
\text{TMP\_DEST}[447:384] & := (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[386:384] \times 64))[63:0]; \\
\text{TMP\_DEST}[511:448] & := (\text{TMP\_SRC2}[511:0] \gg (\text{SRC1}[450:448] \times 64))[63:0];
\end{align*}
\]

F;i;

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP\_DEST[i+63:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0 ;zeroing-masking
    FI;
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

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

\[
\begin{align*}
\text{DEST}[63:0] & := (\text{SRC}[255:0] \gg (\text{IMM8}[1:0] \times 64))[63:0]; \\
\text{DEST}[127:64] & := (\text{SRC}[255:0] \gg (\text{IMM8}[3:2] \times 64))[63:0]; \\
\text{DEST}[191:128] & := (\text{SRC}[255:0] \gg (\text{IMM8}[5:4] \times 64))[63:0]; \\
\text{DEST}[255:192] & := (\text{SRC}[255:0] \gg (\text{IMM8}[7:6] \times 64))[63:0]; \\
\text{DEST}[MAXVL-1:256] & := 0
\end{align*}
\]

Intel C/C++ Compiler Intrinsic Equivalent

\[
\begin{align*}
\text{VPERMQ \_mm512i \_mm512\_permux\_epi64}(\text{\_mm512i a, int imm}); \\
\text{VPERMQ \_mm512i \_mm512\_mask\_permux\_epi64}(\text{\_mm512i s, \_mmask8 k, \_mm512i a, int imm}); \\
\text{VPERMQ \_mm512i \_mm512\_maskz\_permux\_epi64}(\text{\_mmask8 k, \_mm512i a, int imm}); \\
\text{VPERMQ \_mm512i \_mm512\_permux\_epi64}(\text{\_mm512i a, \_mm512i b}); \\
\text{VPERMQ \_mm512i \_mm512\_mask\_permux\_epi64}(\text{\_mmask8 k, \_mm512i a, \_mm512i b}); \\
\text{VPERMQ \_mm512i \_mm512\_maskz\_permux\_epi64}(\text{\_mmask8 k, \_mm512i a, \_mm512i b}); \\
\text{VPERMQ \_mm256i \_mm256\_permux\_epi64}(\text{\_mm256i a, int imm}); \\
\text{VPERMQ \_mm256i \_mm256\_mask\_permux\_epi64}(\text{\_mm256i s, \_mmask8 k, \_mm256i a, int imm}); \\
\text{VPERMQ \_mm256i \_mm256\_maskz\_permux\_epi64}(\text{\_mmask8 k, \_mm256i a, int imm}); \\
\text{VPERMQ \_mm256i \_mm256\_permux\_epi64}(\text{\_mm256i a, \_mm256i b}); \\
\text{VPERMQ \_mm256i \_mm256\_mask\_permux\_epi64}(\text{\_mmask8 k, \_mm256i a, \_mm256i b}); \\
\text{VPERMQ \_mm256i \_mm256\_maskz\_permux\_epi64}(\text{\_mmask8 k, \_mm256i a, \_mm256i b});
\end{align*}
\]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions"; additionally:

#UD
  If VEX.L = 0.
  If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Table 2-50, "Type E4NF Class Exception Conditions"; additionally:

#UD
  If encoded with EVEX.128.
  If VEX.vvvv != 1111B and with imm8.
VPERMT2B—Full Permute of Bytes from Two Tables Overwriting a Table

<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 7D /r VPERMT2B xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in xmm3/m128 and xmm1 using byte indexes in xmm2 and store the byte results in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7D /r VPERMT2B ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>AVX512VL AVX512_VBMI</td>
<td>Permute bytes in ymm3/m256 and ymm1 using byte indexes in ymm2 and store the byte results in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7D /r VPERMT2B zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A V/V</td>
<td>AVX512_VBMI</td>
<td>Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full Mem</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**

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

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

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

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.
Operation
VPERMT2B (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128:
   id := 3;
ELSE IF VL = 256:
   id := 4;
ELSE IF VL = 512:
   id := 5;
FI;
TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
   off := 8*SRC1[j*8 + id: j*8] ;
   IF k1[j] OR *no writemask*:
      DEST[j*8 + 7: j*8] := SRC1[j*8+id+1]? SRC2[off+7:off] : TMP_DEST[off+7:off];
   ELSE IF *zeroing-masking*
      DEST[j*8 + 7: j*8] := 0;
   *ELSE
      DEST[j*8 + 7: j*8] remains unchanged*
   FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;

Intel C/C++ Compiler Intrinsic Equivalent
VPERMT2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMT2B __m512i _mm512_mask_permutex2var_epi8(__m512i a, __mmask64 k, __m512i idx, __m512i b);
VPERMT2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMT2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMT2B __m256i _mm256_mask_permutex2var_epi8(__m256i a, __mmask32 k, __m256i idx, __m256i b);
VPERMT2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMT2B __m128i _mm_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMT2B __m128i _mm_mask_permutex2var_epi8(__m128i a, __mmask16 k, __m128i idx, __m128i b);
VPERMT2B __m128i _mm_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions". 
### VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table

<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 7D /r VPERMT2W xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7D /r VPERMT2W ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7D /r VPERMT2W zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7E /r VPERMT2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7E /r VPERMT2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7E /r VPERMT2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 7E /r VPERMT2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7E /r VPERMT2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7E /r VPERMT2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7F /r VPERMT2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7F /r VPERMT2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7F /r VPERMT2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 7F /r VPERMT2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7F /r VPERMT2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7F /r VPERMT2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>
Description
Permutates 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

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

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands. If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

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

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

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

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

Operation
VPERMT2W (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    id := 2
    Fi;
IF VL = 256
    id := 3
    Fi;
IF VL = 512
    id := 4
    Fi;
    TMP_DEST := DEST
FOR j := 0 TO KL-1
    i := j * 16
    off := 16*SRC1[i+id:j]
    IF k1[j] OR *no writemask*
        THEN
        DEST[i+15:j]=SRC1[i+id+1] ? SRC2[off+15:off]
        : TMP_DEST[off+15:off]
        ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:j] remains unchanged*
        ELSE ; zeroing-masking

VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table
Vol. 2C  5-393
VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table

DEST[i+15:i] := 0
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMT2D/VPERMT2PS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
  id := 1
FI;
IF VL = 256
  id := 2
FI;
IF VL = 512
  id := 3
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
  i := j * 32
  off := 32*SRC1[i+id:i]
  IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN
      DEST[i+31:i] := SRC1[i+id+1] ? SRC2[31:0]
                      : TMP_DEST[off+31:off]
    ELSE
                      : TMP_DEST[off+31:off]
    FI
  ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMT2Q/VPERMT2PD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8 512)
IF VL = 128
  id := 0
FI;
IF VL = 256
  id := 1
FI;
IF VL = 512
  id := 2
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
i := j * 64  
off := 64*SRC1[i+id]  
IF k1[] OR *no writemask*  
THEN  
    IF (EVEX.b = 1) AND (SRC2 *is memory*)  
        THEN  
            DEST[i+63:j] := SRC1[i+i+1] ? SRC2[i63:0]  
                : TMP_DEST[off+63:off]  
        ELSE  
                : TMP_DEST[off+63:off]  
    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[MAXVL-1:VL] := 0  

Intel C/C++ Compiler Intrinsic Equivalent  
VPERMT2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);  
VPERMT2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);  
VPERMT2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);  
VPERMT2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);  
VPERMT2D __m256d _mm256d_permute2var_pd(__m256d a, __m256d idx, __m256d b);  
VPERMT2D __m256d _mm256d_mask_permute2var_pd(__m256d a, __mmask8 k, __m256d idx, __m256d b);  
VPERMT2D __m256d _mm256d_mask2_permute2var_pd(__m256d a, __m256d idx, __mmask8 k, __m256d b);  
VPERMT2D __m256d _mm256d_maskz_permute2var_pd(__mmask8 k, __m256d a, __m256d idx, __m256d b);  
VPERMT2D __m128d _mm128d_permute2var_pd(__m128d a, __m128d idx, __m128d b);  
VPERMT2D __m128d _mm128d_mask_permute2var_pd(__m128d a, __mmask8 k, __m128d idx, __m128d b);  
VPERMT2D __m128d _mm128d_mask2_permute2var_pd(__m128d a, __m128d idx, __mmask8 k, __m128d b);  
VPERMT2D __m128d _mm128d_maskz_permute2var_pd(__mmask8 k, __m128d a, __m128d idx, __m128d b);  
VPERMT2D __m512 _mm512_permute2var_ps(__m512 a, __m512 idx, __m512 b);  
VPERMT2D __m512 _mm512_mask_permute2var_ps(__m512 a, __mmask16 k, __m512 idx, __m512 b);  
VPERMT2D __m512 _mm512_mask2_permute2var_ps(__m512 a, __m512 idx, __mmask16 k, __m512 b);  
VPERMT2D __m512 _mm512_maskz_permute2var_ps(__mmask16 k, __m512 a, __m512 idx, __m512 b);  
VPERMT2Sw/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table  
Vol. 2C 5-395
VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table

INSTRUCTION SET REFERENCE, V-Z

5-396 Vol. 2C

VPERMT2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMT2PS __m256 __mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMT2PS __m256 __mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMT2PS __m128 __mm128_permute2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMT2PS __m128 __mm128_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMT2PS __m128 __mm128_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMT2PS __m128 __mm128_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);

VPERMT2Q __m512i __mm512_permute2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMT2Q __m512i __mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMT2Q __m512i __mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMT2Q __m512i __mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);

VPERMT2W __m512i __mm512_permute2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMT2W __m512i __mm512_mask_permutex2var_epi16(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMT2W __m512i __mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMT2W __m512i __mm512_maskz_permutex2var_epi16(__mmask16 k, __m512i a, __m512i idx, __m512i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPERMT2D/Q/PS/PD: See Table 2-50, “Type E4NF Class Exception Conditions”.
VPERMT2W: See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions”.

5-396 Vol. 2C
VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table
VPEXPANDB/VPEXPANDW — Expand Byte/Word 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 62 /r VPEXPANDB xmm1{k1}{z}, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 128 bits of packed byte values from m128 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 62 /r VPEXPANDB xmm1{k1}{z}, xmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 256 bits of packed byte values from m256 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 62 /r VPEXPANDB ymm1{k1}{z}, ymm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 512 bits of packed byte values from m512 to zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 62 /r VPEXPANDB zmm1{k1}{z}, zmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 128 bits of packed word values from m128 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 62 /r VPEXPANDW xmm1{k1}{z}, xmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 256 bits of packed word values from m256 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 62 /r VPEXPANDW ymm1{k1}{z}, ymm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 512 bits of packed word values from m512 to zmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 62 /r VPEXPANDW zmm1{k1}{z}, zmm2</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Expands up to 512 bits of packed byte integer values from zmm2 to zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

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

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Moves 128, 256 or 512 bits of packed byte integer values from the source operand (memory operand) to the destination operand (register operand). This instruction is used to load from an int8 vector register or memory location while inserting the data into sparse elements of destination vector register using the active elements pointed out by the operand writemask.

This instruction supports memory fault suppression.

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

**Operation**

**VPExPANDB**

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

k := 0

FOR j := 0 TO KL-1:

IF k1[j] OR *no writemask*:

  DEST.byte[j] := SRC.byte[k];
  k := k + 1

ELSE:

  IF *merging-masking*:

    *DEST.byte[j] remains unchanged*

  ELSE:  ; zeroing-masking

    DEST.byte[j] := 0

DEST[MAX_VL-1:VL] := 0

**VPExPANDw**

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

k := 0

FOR j := 0 TO KL-1:

IF k1[j] OR *no writemask*:

  DEST.word[j] := SRC.word[k];
  k := k + 1

ELSE:

  IF *merging-masking*:

    *DEST.word[j] remains unchanged*

  ELSE:  ; zeroing-masking

    DEST.word[j] := 0

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

VPEXPAND __m128i _mm_mask_expand_epi8(__m128i, __mmask16, __m128i);
VPEXPAND __m128i _mm_maskz_expand_epi8(__mmask16, __m128i);
VPEXPAND __m128i _mm_mask Expandloadu_epi8(__m128i, __mmask16, const void*);
VPEXPAND __m256i _mm256_mask Expand_epi8(__m256i, __mmask32, __m256i);
VPEXPAND __m256i _mm256_maskzExpand_epi8(__mmask32, __m256i);
VPEXPAND __m256i _mm256_mask expandloadu_epi8(__m256i, __mmask32, const void*);
VPEXPAND __m512i _mm512_mask Expand_epi8(__m512i, __mmask64, __m512i);
VPEXPAND __m512i _mm512_maskz expand_epi8(__mmask64, __m512i);
VPEXPAND __m512i _mm512_mask expandloadu_epi8(__m512i, __mmask64, const void*);
VPEXPAND __m128i _mm_mask Expand_epi16(__m128i, __mmask8, __m128i);
VPEXPAND __m128i _mm_maskz expand_epi16(__mmask8, __m128i);
VPEXPAND __m256i _mm256_mask expand_epi16(__m256i, __mmask16, __m256i);
VPEXPAND __m256i _mm256_maskz expand_epi16(__mmask16, __m256i);
VPEXPAND __m512i _mm512_mask expand_epi16(__m512i, __mmask32, __m512i);
VPEXPAND __m512i _mm512_maskz expand_epi16(__mmask32, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions".
**VPEXPANDD**—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register

### 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.128.66.0F38.W0 89 /r VPEXPANDD xmm1 {k1}{z}, xmm2/m128</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 89 /r VPEXPANDD ymm1 {k1}{z}, ymm2/m256</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 89 /r VPEXPANDD zmm1 {k1}{z}, zmm2/m512</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</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[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDD __m512i _mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_maskz_expandloadu_epi32(__mmask16 k, void * a);
VPEXPANDD __m512i _mm512_mask Expand_epi32(__m512i s, __mmask16 k, __m512i a);
VPEXPANDD __m512i _mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
VPEXPANDD __m256i _mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m256i _mm256_mask expand_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i _mm256_maskz expand_epi32(__mmask8 k, __m256i a);
VPEXPANDD __m128i _mm_mask expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i _mm_maskz expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m128i _mm_mask expand_epi32(__m128i s, __mmask8 k, __m128i a);
VPEXPANDD __m128i _mm_maskz expand_epi32(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”;
additionally:

#UD If EVEX.vvvv != 1111B.
VPEXPANDQ—Load Sparse Packed Quadword Integer Values from Dense Memory / Register

### Instruction Set Reference, V-Z

#### 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 89 /r</td>
<td>A</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>VPEXPANDQ xmm1 {k1}[z], xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 89 /r</td>
<td>A</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>VPEXPANDQ ymm1 {k1}[z], ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 89 /r</td>
<td>A</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>
<tr>
<td>VPEXPANDQ 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>Tuple Type</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>Tuple1 Scalar</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** \(KL-1\)

\(i := j \times 64\)

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" \(;"zeroing-masking"\) ELSE \("DEST[i+63:j] := 0"\)

\(Fi\)

ENDFOR

\[DEST[MAXVL-1:VL] := 0\]
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_maskz_expand_epi64(___m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_maskz_expandloadu_epi64( __mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_maskz_expand_epi64(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskz_expand_epi64( __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskz_expand_epi64(___m256i s, __mmask8 k, __m256i a);
VPEXPANDQ __m128i _mm_maskz_expandloadu_epi64( __m256i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm_maskz_expand_epi64(___m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm128_maskz_expand_epi64( __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm128_maskz Expand_epi64( __m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm128_maskz expand_epi64( __m128i s, __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”;
additionally:

#UD If EVEX.vvvv != 1111B.
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.128.66.0F38.W0 90 /r VPGATHERDD xmm1, vm32x, xmm2</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.128.66.0F38.W0 91 /r VPGATHERQD xmm1, vm64x, xmm2</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.256.66.0F38.W0 90 /r VPGATHERDD ymm1, vm32y, ymm2</td>
<td>RMV</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using dword indices specified in vm32y, gather dword from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 91 /r VPGATHERQD xmm1, vm64y, xmm2</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 xmm2. Conditionally gathered elements are merged into xmm1.</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>RMV</td>
<td>ModRMreg (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 zero the upper 64 bits of the destination.
VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zero 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;

VPGATHERDD (VEX.128 version)
MASK[MAXVL-1:128] := 0;
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
FOR j := 0 to 3
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;
VPGATHERQD (VEX.128 version)
MASK[MAXVL-1:64] := 0;
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
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
DEST[MAXVL-1:64] := 0;

VPGATHERDD (VEX.256 version)
MASK[MAXVL-1:256] := 0;
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
DEST[MAXVL-1:256] := 0;
VPGATHERQD (VEX.256 version)
MASK[MAXVL-1:128] := 0;
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
DEST[MAXVL-1:128] := 0;

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, __m256i mask, const int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Table 2-27, "Type 12 Class Exception Conditions".
VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword with Signed Dword Indices

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.

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB: VSIBindex</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
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 VINDEX. The instruction will #UD fault if the k0 mask register is specified.
The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**
BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1 or 4 byte displacement

**VPGATHERDD (EVEX encoded version)**
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j]
    THEN DEST[i+31:i] := MEM[BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
    k1[j] := 0
    ELSE *DEST[i+31:i] := remains unchanged* ; Only merging masking is allowed
  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0

**VPGATHERDQ (EVEX encoded version)**
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j]
    THEN DEST[i+63:i] :=
      MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
    k1[j] := 0
    ELSE *DEST[i+63:i] := remains unchanged* ; Only merging masking is allowed
  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD __m512i __m512i __m512i __m512i vdx, void * base, int scale);
VPGATHERDD __m512i __m512i __m512i __m512i vdx, void * base, int scale);
VPGATHERDD __m256i __m256i __m512i __m256i vdx, void * base, int scale);
VPGATHERDD __m128i __m128i __m256i __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-61, “Type E12 Class Exception Conditions”.
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.128.66.0F38.W1 90 /r</td>
<td>A</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.128.66.0F38.W1 91 /r</td>
<td>A</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.256.66.0F38.W1 90 /r</td>
<td>A</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 ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 91 /r</td>
<td>A</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 ymm2. Conditionally gathered elements are merged into ymm1.</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>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 and 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)
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 1
i := j * 64;
IF MASK[63+i] THEN
Maska := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
ELSE
Maska := 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
DEST[MAXVL-1:128] := 0;
VPGATHERQQ (VEX.128 version)
MASK[MAXVL-1:128] := 0;
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
DEST[MAXVL-1:128] := 0;

VPGATHERQQ (VEX.256 version)
MASK[MAXVL-1:256] := 0;
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
DEST[MAXVL-1:256] := 0;

VPGATHERDQ (VEX.256 version)
MASK[MAXVL-1:256] := 0;
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[i+63] THEN
  DEST[i+63] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +63] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;

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

VPGATHERDQ: __m128i _mm_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);

VPGATHERDQ: __m128i _mm_mask_i32gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);

VPGATHERDQ: __m256i _mm256_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);

VPGATHERDQ: __m256i _mm256_mask_i32gather_epi64 (__m256i src, __int64 const * base, __m128i index, __m256i mask, const int scale);

VPGATHERQQ: __m128i _mm_i64gather_epi64 (__int64 const * base, __m128i index, const int scale);

VPGATHERQQ: __m128i _mm_mask_i64gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);

VPGATHERQQ: __m256i _mm256_i64gather_epi64 (__int64 const * base, __m256i index, const int scale);

VPGATHERQQ: __m256i _mm256_mask_i64gather_epi64 (__m256i src, __int64 const * base, __m256i index, __m256i mask, const int scale);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 2-27, “Type 12 Class Exception Conditions”.
**VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword 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 91 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>VPGATHERQD xmm1 {k1}, vm64x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 91 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>VPGATHERQD xmm1 {k1}, vm64y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 91 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>VPGATHERQD ymm1 {k1}, vm64z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 91 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>VPGATHERQD xmm1 {k1}, vm64x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 91 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>VPGATHERQD ymm1 {k1}, vm64y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 91 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
</tr>
<tr>
<td>VPGATHERQD zmm1 {k1}, vm64z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

**Operation**

BASE_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1 or 4 byte displacement

**VPGATHERQD (EVEX encoded version)**

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

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j]
    THEN DEST[i+31:i] := MEM[B\_ADDR + (VINDEX[i+63:k]) * SCALE + DISP]
    k1[j] := 0
    ELSE *DEST[i+31:i] := remains unchanged* ; Only merging masking is allowed
  FI;
ENDFOR
k1[MAX\_KL-1:KL] := 0
DEST[MAXVL-1:VL/2] := 0

**VPGATHERQQ (EVEX encoded version)**

KL, VL = (2, 64), (4, 128), (8, 256)

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j]
    THEN DEST[i+63:i] :=
      MEM[B\_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
    k1[j] := 0
    ELSE *DEST[i+63:i] := remains unchanged* ; Only merging masking is allowed
  FI;
ENDFOR
k1[MAX\_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent
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 __m256i __mm_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Table 2-61, “Type E12 Class Exception Conditions”.
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 Flag</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>A</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>A</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>A</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>A</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>A</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>A</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>Tuple Type</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>Full</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 a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

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

VPLZCNTD

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j*32
  IF MaskBit(j) OR *no writemask*
    THEN
      temp := 32
      DEST[i+31:i] := 0
      WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
        DO
          temp := temp – 1
          DEST[i+31:i] := DEST[i+31:i] + 1
        OD
    ELSE
      IF *merging-masking*
        THEN *DEST[i+31:i] remains unchanged*
      ELSE DEST[i+31:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

VPLZCNTQ

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j*64
  IF MaskBit(j) OR *no writemask*
    THEN
      temp := 64
      DEST[i+63:i] := 0
      WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
        DO
          temp := temp – 1
          DEST[i+63:i] := DEST[i+63:i] + 1
        OD
    ELSE
      IF *merging-masking*
        THEN *DEST[i+63:i] remains unchanged*
      ELSE DEST[i+63:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VPLZCNTD __m512i _mm512_lzcnt_epi32(__m512i a);
VPLZCNTD __m512i _mm512_mask_lzcnt_epi32(__m512i s, __mmask16 m, __m512i a);
VPLZCNTD __m512i _mm512_maskz_lzcnt_epi32(__mmask16 m, __m512i a);
VPLZCNTQ __m512i _mm512_lzcnt_epi64(__m512i a);
VPLZCNTQ __m512i _mm512_mask_lzcnt_epi64(__m512i s, __mmask8 m, __m512i a);
VPLZCNTQ __m512i _mm512_maskz_lzcnt_epi64(__mmask8 m, __m512i a);
VPLZCNTD __m256i _mm256_lzcnt_epi32(__m256i a);
VPLZCNTD __m256i _mm256_mask_lzcnt_epi32(__m256i s, __mmask8 m, __m256i a);
VPLZCNTD __m256i _mm256_maskz_lzcnt_epi32(__mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_lzcnt_epi64(__m256i a);
VPLZCNTQ __m256i _mm256_mask_lzcnt_epi64(__m256i s, __mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_maskz_lzcnt_epi64(__mmask8 m, __m256i a);
VPLZCNTD __m128i _mm_lzcnt_epi32(__m128i a);
VPLZCNTD __m128i _mm_mask_lzcnt_epi32(__m128i s, __mmask8 m, __m128i a);
VPLZCNTD __m128i _mm_maskz_lzcnt_epi32(__mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_lzcnt_epi64(__m128i a);
VPLZCNTQ __m128i _mm_mask_lzcnt_epi64(__m128i s, __mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_maskz_lzcnt_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions”. 
VPMADD52HUQ—Packed Multiply of Unsigned 52-bit Unsigned Integers and Add High 52-bit Products to 64-bit Accumulators

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>32/64 bit Mode</th>
<th>CPUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>A V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>A V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>A V/V</td>
<td>AVX512_IFMA</td>
<td>Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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

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

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

VPMADD52HUQ (EVEX encoded)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64;
  IF k1[j] OR *no writemask* THEN
    IF src2 is Memory AND EVEX.b=1 THEN
      tsrc2[63:0] := ZeroExtend64(src2[51:0]);
    ELSE
      tsrc2[63:0] := ZeroExtend64(src2[i+51:i+52]);
    FI;
    Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];
    Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[103:52]);
    DEST[i+63:i] := Temp2[63:0];
  ELSE
    IF *zeroing-masking* THEN
      DEST[i+63:i] := 0;
    ELSE *merge-masking*
      DEST[i+63:i] is unchanged;
    FI;
  FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPMADD52HUQ __m512i _mm512_madd52hi_epu64( __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_mask_madd52hi_epu64(__m512i s, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_maskz_madd52hi_epu64( __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m256i _mm256_madd52hi_epu64( __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_mask_madd52hi_epu64(__m256i s, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_maskz_madd52hi_epu64( __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m128i _mm_madd52hi_epu64( __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_mask_madd52hi_epu64(__m128i s, __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm_maskz_madd52hi_epu64( __mmask8 k, __m128i a, __m128i b, __m128i c);

Flags Affected

None.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions”.
**VPMADD52LUQ—Packed Multiply of Unsigned 52-bit Integers and Add the Low 52-bit Products to Qword Accumulators**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>32/64 bit Mode Support</th>
<th>CPUID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 B4 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 B4 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA AVX512VL</td>
<td>Multiply unsigned 52-bit integers in ymm2 and ymm3/m256 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 B4 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_IFMA</td>
<td>Multiply unsigned 52-bit integers in zmm2 and zmm3/m512 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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**

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

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

**VPMADD52LUQ (EVEX encoded)**

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

FOR \(j := 0 \) TO \(KL-1\)

\[i := j \times 64;\]

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

IF src2 is Memory AND EVEX.b=1 \ THEN

\[tsrc2[63:0] := \text{ZeroExtend64}(\text{src2}[51:0]);\]

ELSE

\[tsrc2[63:0] := \text{ZeroExtend64}(\text{src2}[i+51:i]);\]

**ENDIF**

\[\text{Temp128}[127:0] := \text{ZeroExtend64}(\text{src1}[i+51:i]) \times \text{tsrc2}[63:0];\]

\[\text{Temp2}[63:0] := \text{DEST}[i+63:i] + \text{ZeroExtend64}(\text{temp128}[51:0]);\]

\[\text{DEST}[i+63:i] := \text{Temp2}[63:0];\]

**ELSE**

IF *zeroing-masking* \ THEN

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

ELSE *merge-masking*

\[\text{DEST}[i+63:i] \text{ is unchanged};\]

**ENDIF**

**ENDFOR**

\[\text{DEST}[\text{MAX}_Vl-1:Vl] := 0;\]

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

- \(\text{VPMADD52LUQ} \_\_m512i \_\_mm512\_madd52lo\_epu64(\_\_m512i \text{a}, \_\_m512i \text{b}, \_\_m512i \text{c});\)
- \(\text{VPMADD52LUQ} \_\_m512i \_\_mm512\_\_mask\_madd52lo\_epu64(\_\_m512i \text{s}, \_\_mmask8 \text{k}, \_\_m512i \text{a}, \_\_m512i \text{b}, \_\_m512i \text{c});\)
- \(\text{VPMADD52LUQ} \_\_m256i \_\_mm256\_madd52lo\_epu64(\_\_m256i \text{a}, \_\_m256i \text{b}, \_\_m256i \text{c});\)
- \(\text{VPMADD52LUQ} \_\_m256i \_\_mm256\_\_mask\_madd52lo\_epu64(\_\_m256i \text{s}, \_\_mmask8 \text{k}, \_\_m256i \text{a}, \_\_m256i \text{b}, \_\_m256i \text{c});\)
- \(\text{VPMADD52LUQ} \_\_m128i \_\_mm128\_madd52lo\_epu64(\_\_m128i \text{a}, \_\_m128i \text{b}, \_\_m128i \text{c});\)
- \(\text{VPMADD52LUQ} \_\_m128i \_\_mm128\_\_mask\_madd52lo\_epu64(\_\_m128i \text{s}, \_\_mmask8 \text{k}, \_\_m128i \text{a}, \_\_m128i \text{b}, \_\_m128i \text{c});\)
- \(\text{VPMADD52LUQ} \_\_m128i \_\_mm128\_\_mask\_z\_madd52lo\_epu64(\_\_mmask8 \text{k}, \_\_m128i \text{a}, \_\_m128i \text{b}, \_\_m128i \text{c});\)

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 2-49, “Type E4 Class Exception Conditions”.

---

5-424  Vol. 2C  VPMADD52LUQ—Packed Multiply of Unsigned 52-bit Integers and Add the Low 52-bit Products to Qword Accumulators
VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores

**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 (r)</td>
<td>ModRMr/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>MVR</td>
<td>ModRMr/m (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (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

\[
\begin{align*}
\text{DEST}[31:0] &= \text{IF (SRC1[31]) Load}_32(\text{mem}) \text{ ELSE 0} \\
\text{DEST}[63:32] &= \text{IF (SRC1[63]) Load}_32(\text{mem} + 4) \text{ ELSE 0} \\
\text{DEST}[95:64] &= \text{IF (SRC1[95]) Load}_32(\text{mem} + 8) \text{ ELSE 0} \\
\text{DEST}[127:96] &= \text{IF (SRC1[127]) Load}_32(\text{mem} + 12) \text{ ELSE 0} \\
\text{DEST}[159:128] &= \text{IF (SRC1[159]) Load}_32(\text{mem} + 16) \text{ ELSE 0} \\
\text{DEST}[191:160] &= \text{IF (SRC1[191]) Load}_32(\text{mem} + 20) \text{ ELSE 0} \\
\text{DEST}[223:192] &= \text{IF (SRC1[223]) Load}_32(\text{mem} + 24) \text{ ELSE 0} \\
\text{DEST}[255:224] &= \text{IF (SRC1[255]) Load}_32(\text{mem} + 28) \text{ ELSE 0}
\end{align*}
\]

**VPMASKMOVD** - 128-bit load

\[
\begin{align*}
\text{DEST}[31:0] &= \text{IF (SRC1[31]) Load}_32(\text{mem}) \text{ ELSE 0} \\
\text{DEST}[63:32] &= \text{IF (SRC1[63]) Load}_32(\text{mem} + 4) \text{ ELSE 0} \\
\text{DEST}[95:64] &= \text{IF (SRC1[95]) Load}_32(\text{mem} + 8) \text{ ELSE 0} \\
\text{DEST}[127:97] &= \text{IF (SRC1[127]) Load}_32(\text{mem} + 12) \text{ ELSE 0} \\
\text{DEST}[\text{MAXVL}-1:128] &= 0
\end{align*}
\]

**VPMASKMOVQ** - 256-bit load

\[
\begin{align*}
\text{DEST}[63:0] &= \text{IF (SRC1[63]) Load}_64(\text{mem}) \text{ ELSE 0} \\
\text{DEST}[127:64] &= \text{IF (SRC1[127]) Load}_64(\text{mem} + 8) \text{ ELSE 0} \\
\text{DEST}[195:128] &= \text{IF (SRC1[191]) Load}_64(\text{mem} + 16) \text{ ELSE 0} \\
\text{DEST}[255:196] &= \text{IF (SRC1[255]) Load}_64(\text{mem} + 24) \text{ ELSE 0}
\end{align*}
\]

**VPMASKMOVQ** - 128-bit load

\[
\begin{align*}
\text{DEST}[63:0] &= \text{IF (SRC1[63]) Load}_64(\text{mem}) \text{ ELSE 0} \\
\text{DEST}[127:64] &= \text{IF (SRC1[127]) Load}_64(\text{mem} + 16) \text{ ELSE 0} \\
\text{DEST}[\text{MAXVL}-1:128] &= 0
\end{align*}
\]

**VPMASKMOVD** - 256-bit store

\[
\begin{align*}
\text{IF (SRC1[31]) DEST}[31:0] &= \text{SRC2}[31:0] \\
\text{IF (SRC1[63]) DEST}[63:32] &= \text{SRC2}[63:32] \\
\text{IF (SRC1[95]) DEST}[95:64] &= \text{SRC2}[95:64] \\
\text{IF (SRC1[127]) DEST}[127:96] &= \text{SRC2}[127:96] \\
\text{IF (SRC1[159]) DEST}[159:128] &= \text{SRC2}[159:128] \\
\text{IF (SRC1[191]) DEST}[191:160] &= \text{SRC2}[191:160] \\
\text{IF (SRC1[223]) DEST}[223:192] &= \text{SRC2}[223:192] \\
\text{IF (SRC1[255]) DEST}[255:224] &= \text{SRC2}[255:224]
\end{align*}
\]
VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores

VPMASKMOVD - 128-bit store
IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]

VPMASKMOVD - 256-bit store
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]

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
VPMASKMOV: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)
VPMASKMOV: 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, __m256i b);
VPMASKMOV: __m128i _mm_maskload_epi32(int const *a, __m128i mask)
VPMASKMOVQ: void _mm_maskstore_epi32(int *a, __m128i mask, __m128i b)
VPMASKMOVQ: __m128i _mm_maskload_epi64(__int const *a, __m128i mask);
VPMASKMOVQ: void _mm_maskstore_epi64(__int const *a, __m128i mask, __m128i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
See Table 2-23, "Type 6 Class Exception Conditions" (No AC# reported for any mask bit combinations).
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</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 29 /r VPMOVB2M k1, ymm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 29 /r VPMOVB2M k1, zmm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 29 /r VPMOVW2M k1, xmm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 29 /r VPMOVW2M k1, ymm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 29 /r VPMOVW2M k1, zmm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 39 /r VPMOVD2M k1, xmm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 39 /r VPMOVD2M k1, ymm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 39 /r VPMOVD2M k1, zmm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 39 /r VPMOVQ2M k1, xmm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 39 /r VPMOVQ2M k1, ymm1</td>
<td>RM</td>
<td>V/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>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 39 /r VPMOVQ2M k1, zmm1</td>
<td>RM</td>
<td>V/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>
</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 __mm_movepi8_mask( __m128i );
VPMPOVD2M __mmask8  __mm_movepi32_mask( __m128i );
VPMPOVQ2M __mmask8  __mm_movepi64_mask( __m128i );
VPMPOVW2M __mmask8  __mm_movepi16_mask( __m128i );

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-55, "Type E7NM Class Exception Conditions"; additionally:
#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 VPMOVDB xmm1/m32 {k1}{z}, xmm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 21 /r VPMOVSDB xmm1/m32 {k1}{z}, xmm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m32 {k1}{z}, xmm2</td>
<td>A 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 unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 31 /r VPMOVDB xmm1/m64 {k1}{z}, ymm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 21 /r VPMOVSDB xmm1/m64 {k1}{z}, ymm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m64 {k1}{z}, ymm2</td>
<td>A 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 unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 31 /r VPMOVDB xmm1/m128 {k1}{z}, zmm2</td>
<td>A V/V</td>
<td>AVX512VL</td>
<td>Converts 16 packed double-word integers from zmm2 into 16 packed byte integers in xmm1/m128 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 21 /r VPMOVSDB xmm1/m128 {k1}{z}, zmm2</td>
<td>A V/V</td>
<td>AVX512VL</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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB xmm1/m128 {k1}{z}, zmm2</td>
<td>A V/V</td>
<td>AVX512VL</td>
<td>Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned byte integers in xmm1/m128 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Quarter Mem</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 (MAXVL-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 \(j := 0\) TO \(KL-1\)

\[i := j \times 8\]
\[m := j \times 32\]

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

THEN \(\text{DEST}[i+7:i] := \text{TruncateDoubleWordToByte}(\text{SRC}[m+31:m])\)

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST*[\(i+7:i\] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\(\text{DEST}[i+7:i] := 0\)

FI

FI;

ENDFOR

\(\text{DEST}[^{MAXVL-1:VL/4}]: = 0;\)

VPMOVDB instruction (EVEX encoded versions) when dest is memory

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

FOR \(j := 0\) TO \(KL-1\)

\[i := j \times 8\]
\[m := j \times 32\]

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

THEN \(\text{DEST}[i+7:i] := \text{TruncateDoubleWordToByte}(\text{SRC}[m+31:m])\)

ELSE *DEST*[\(i+7:i\] remains unchanged* ; merging-masking

FI;

ENDFOR

VPMOVDB instruction (EVEX encoded versions) when dest is a register

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

FOR \(j := 0\) TO \(KL-1\)

\[i := j \times 8\]
\[m := j \times 32\]

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

THEN \(\text{DEST}[i+7:i] := \text{SaturateSignedDoubleWordToByte}(\text{SRC}[m+31:m])\)

ELSE *merging-masking* ; merging-masking

THEN *DEST*[\(i+7:i\] remains unchanged*

ELSE *zeroing-masking* ; zeroing-masking

\(\text{DEST}[i+7:i] := 0\)

FI

FI;

ENDFOR

\(\text{DEST}[^{MAXVL-1:VL/4}]: = 0;\)
VPMOVVSDB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateSignedDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VPMOVUSDB instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
    ELSE *merging-masking*
    THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking*
        DEST[i+7:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;

VPMOVUSDB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m512i s, __mmask16 k, __m512i a);
VPMOVDB void __m512_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVDSDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDSDB __m128i _mm512_mask_cvtepi32_epi8(__m512i s, __mmask16 k, __m512i a);
VPMOVDSDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDSDB void __m512_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVUSDB __m128i _mm512_mask_cvtepi32_epi8(__m512i s, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVUSDB void __m512_storeu_epi8(void * d, __mmask16 k, __m512i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions”; additionally:

#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>A</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>A</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>A</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>A</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>A</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>A</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>A</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>A</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>A</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>

**Opcode/Instruction**

- **Op/En Tuple Type**: Half Mem
- **Operand 1**: ModRM:r/m (w)
- **Operand 2**: ModRM:reg (r)
- **Operand 3**: NA
- **Operand 4**: NA

**Description**

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW converts unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

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

VPMOVDw instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged* ; zeroing-masking
      ELSE *DEST[i+15:i] := 0*
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL/2] := 0;

VPMOVDw instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI
ENDFOR

VPMOVSDw instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateSignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged* ; zeroing-masking
      ELSE *DEST[i+15:i] := 0*
    FI
  FI
ENDFOR
DEST[MAXVL-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[MAXVL-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( __m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i __mm512_maskz_cvtepi32_epi16( __mmask16 k, __m512i a);
VPMOVDW void __mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVDW __m256i __mm512_cvtepi32_epi16( __m512i a);
VPMOVDW __m256i __mm512_mask_cvtepi32_epi16( __m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i __mm512_maskz_cvtepi32_epi16( __mmask16 k, __m512i a);
VPMOVDW void __mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVDW __m512i __mm256_cvtepi32_epi16( __m256i a);
VPMOVDW __m512i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask8 k, __m256i a);
VPMOVDW __m512i __mm256_maskz_cvtepi32_epi16( __mmask8 k, __m256i a);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m256i a);
VPMOVDW __m512i __mm256_cvtepi32_epi16( __m256i a);
VPMOVDW __m512i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask16 k, __m512i a);
VPMOVDW __m512i __mm256_maskz_cvtepi32_epi16( __mmask16 k, __m512i a);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVDW __m128i __mm256_mask_cvtepi32_epi16( __m256i a);
VPMOVDW __m128i __mm256_maskz_cvtepi32_epi16( __mmask8 k, __m256i a);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m256i a);
VPMOVDW __m128i __mm256_cvtepi32_epi16( __m256i a);
VPMOVDW __m128i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_maskz_cvtepi32_epi16( __mmask8 k, __m256i b);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_cvtepi32_epi16( __m256i a);
VPMOVDW __m128i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_maskz_cvtepi32_epi16( __mmask8 k, __m256i b);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_cvtepi32_epi16( __m256i a);
VPMOVDW __m128i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_maskz_cvtepi32_epi16( __mmask8 k, __m256i b);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_cvtepi32_epi16( __m256i a);
VPMOVDW __m128i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_maskz_cvtepi32_epi16( __mmask8 k, __m256i b);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_cvtepi32_epi16( __m256i a);
VPMOVDW __m128i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_maskz_cvtepi32_epi16( __mmask8 k, __m256i b);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_cvtepi32_epi16( __m256i a);
VPMOVDW __m128i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_maskz_cvtepi32_epi16( __mmask8 k, __m256i b);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void * d, __mmask8 k, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-53, ”Type E6 Class Exception Conditions”; additionally:

#UD If EVEX.vvvv != 1111B.
VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

**Instruction**

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.**

### 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.F3.0F38.W0 2B /r VPMOVM2B xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</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>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 2B /r VPMOVM2B ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</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>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 2B /r VPMOVM2B zmm1, k1</td>
<td>RM</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 2B /r VPMOVM2W xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</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>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 2B /r VPMOVM2W ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512Bw</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>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 2B /r VPMOVM2W zmm1, k1</td>
<td>RM</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 3B /r VPMOVM2D xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</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>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 3B /r VPMOVM2D ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</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>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 3B /r VPMOVM2D zmm1, k1</td>
<td>RM</td>
<td>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>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 3B /r VPMOVM2Q xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</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>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 3B /r VPMOVM2Q ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</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>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 3B /r VPMOVM2Q zmm1, k1</td>
<td>RM</td>
<td>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>
</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)**
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
i := j * 8
IF SRC[j]
THEN DEST[i+7:j] := -1
ELSE DEST[i+7:j] := 0
Fi;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VPMOVM2W (EVEX encoded versions)**
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
i := j * 16
IF SRC[j]
THEN DEST[i+15:j] := -1
ELSE DEST[i+15:j] := 0
Fi;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VPMOVM2D (EVEX encoded versions)**
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
i := j * 32
IF SRC[j]
THEN DEST[i+31:j] := -1
ELSE DEST[i+31:j] := 0
Fi;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VPMOVM2Q (EVEX encoded versions)**
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF SRC[j]
THEN DEST[i+63:j] := -1
ELSE DEST[i+63:j] := 0
Fi;
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVM2B __m512i __mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i __mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i __mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i __mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i __mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i __mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i __mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i __mm256_movm_epi16(__mmask8);
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 Table 2-55, "Type E7NM Class Exception Conditions"; additionally:

#UD If EVEX.vvvv != 1111B.
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>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 22 /r VPMOVSQB xmm1/m16 {k1}{z}, xmm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m16 {k1}{z}, xmm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 32 /r VPMOVQB xmm1/m32 {k1}{z}, ymm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 22 /r VPMOVSQB xmm1/m32 {k1}{z}, ymm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m32 {k1}{z}, ymm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 32 /r VPMOVQB xmm1/m64 {k1}{z}, zmm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 22 /r VPMOVSQB xmm1/m64 {k1}{z}, zmm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB xmm1/m64 {k1}{z}, zmm2</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Eighth Mem</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 (MAXVL-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 \(j := 0 \) TO \(KL-1\)
  \(i := j \times 8\)
  \(m := j \times 64\)
  IF \(k1[j] \) OR *no writemask*
    THEN \(DEST[i+7:i] := \text{TruncateQuadWordToByte} (SRC[m+63:m])\)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged* 
      ELSE *zeroing-masking* ; zeroing-masking
        \(DEST[i+7:i] := 0\)
    FI
  FI;
ENDFOR
\(DEST[\text{MAXVL-1:VL/8}] := 0;\)

VPMOVQB instruction (EVEX encoded versions) when dest is memory

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

FOR \(j := 0 \) TO \(KL-1\)
  \(i := j \times 8\)
  \(m := j \times 64\)
  IF \(k1[j] \) OR *no writemask*
    THEN \(DEST[i+7:i] := \text{TruncateQuadWordToByte} (SRC[m+63:m])\)
  ELSE
    *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVSQB instruction (EVEX encoded versions) when dest is a register

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

FOR \(j := 0 \) TO \(KL-1\)
  \(i := j \times 8\)
  \(m := j \times 64\)
  IF \(k1[j] \) OR *no writemask*
    THEN \(DEST[i+7:i] := \text{SaturateSignedQuadWordToByte} (SRC[m+63:m])\)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged* 
      ELSE *zeroing-masking* ; zeroing-masking
        \(DEST[i+7:i] := 0\)
    FI
  FI;
ENDFOR
\(DEST[\text{MAXVL-1:VL/8}] := 0;\)
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[MAXVL-1:VL/8] := 0;

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedQuadWordToByte (SRC[m+63:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
**Intel C/C++ Compiler Intrinsic Equivalents**

VPMOVQB __m128i _mm512_cvtepi64_epi8( __m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8( __m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512Mask_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8( void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8( __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8( void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512Maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB void _mm512_maskz_cvtepi64_storeu_epi8( void * d, __mmask8 k, __m512i a);

VPMOVQB __m128i _mm512_cvtsepi64_epi8( __m512i a);
VPMOVQB __m128i _mm512_mask_cvtsepi64_epi8( __m512i a, __mmask8 k, __m512i b);
VPMOVQB __m128i _mm512Mask_cvtsepi64_epi8( __m512i a, __mmask8 k, __m512i b);
VPMOVQB __m128i _mm512_maskz_cvtsepi64_epi8( __mmask8 k, __m512i b);
VPMOVQB void _mm512_mask_cvtsepi64_storeu_epi8( void * d, __mmask8 k, __m512i b);
VPMOVQB __m128i _mm_cvtsepi64_epi8( __m128i a);
VPMOVQB __m128i _mm_mask_cvtsepi64_epi8( __m128i a, __mmask8 k, __m128i b);
VPMOVQB __m128i _mm_maskz_cvtsepi64_epi8( __mmask8 k, __m128i b);
VPMOVQB void _mm_mask_cvtsepi64_storeu_epi8( void * d, __mmask8 k, __m128i b);

VPMOVQB __m128i _mm256_cvtepi64_epi8( __m256i a);
VPMOVQB __m128i _mm256_mask_cvtepi64_epi8( __m128i a, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm256Mask_cvtepi64_epi8( __m128i a, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm256_maskz_cvtepi64_epi8( __mmask8 k, __m256i b);
VPMOVQB void _mm256_mask_cvtepi64_storeu_epi8( void * d, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm256_cvtsepi64_epi8( __m256i a);
VPMOVQB __m128i _mm256_mask_cvtsepi64_epi8( __m128i a, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm256Mask_cvtsepi64_epi8( __m128i a, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm256_maskz_cvtsepi64_epi8( __mmask8 k, __m256i b);
VPMOVQB void _mm256_mask_cvtsepi64_storeu_epi8( void * d, __mmask8 k, __m256i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions”; additionally:

#UD  If EVEX.vvvv != 1111B.
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

**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 double-words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

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

---

**Instruction Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Half Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</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.128.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 [k1][z], xmm2</td>
<td>A</td>
<td>V/V</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 VPMOVSQD xmm1/m64 [k1][z], xmm2</td>
<td>A</td>
<td>V/V</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 VPMOVUSQD xmm1/m64 [k1][z], xmm2</td>
<td>A</td>
<td>V/V</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 unsigned saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 35 /r VPMOVQD xmm1/m128 [k1][z], ymm2</td>
<td>A</td>
<td>V/V</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 VPMOVSQD xmm1/m128 [k1][z], ymm2</td>
<td>A</td>
<td>V/V</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 VPMOVUSQD xmm1/m128 [k1][z], ymm2</td>
<td>A</td>
<td>V/V</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 unsigned saturation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 35 /r VPMOVQD ymm1/m256 [k1][z], zmm2</td>
<td>A</td>
<td>V/V</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 VPMOVSQD ymm1/m256 [k1][z], zmm2</td>
<td>A</td>
<td>V/V</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 VPMOVUSQD ymm1/m256 [k1][z], zmm2</td>
<td>A</td>
<td>V/V</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 unsigned saturation subject to writemask k1.</td>
</tr>
</tbody>
</table>
**Operation**

**VPMOVQD instruction (EVEX encoded version) reg-reg form**

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

FOR \(j := 0\) TO \(KL-1\)

\[i := j * 32\]
\[m := j * 64\]

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

\[\text{THEN } \text{DEST}[i+31:i] := \text{TruncateQuadWordToDWord}(\text{SRC}[m+63:m])\]
\[\text{ELSE } \text{DEST}[i+31:i] := 0\]

\[\text{FI}\]

\[\text{FI;}\]

ENDFOR

\[\text{DEST}[\text{MAXVL-1:VL/2}] := 0;\]

**VPMOVQD instruction (EVEX encoded version) memory form**

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

FOR \(j := 0\) TO \(KL-1\)

\[i := j * 32\]
\[m := j * 64\]

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

\[\text{THEN } \text{DEST}[i+31:i] := \text{TruncateQuadWordToDWord}(\text{SRC}[m+63:m])\]
\[\text{ELSE } \text{DEST}[i+31:i] \text{ remains unchanged}\]

\[\text{FI;}\]

ENDFOR

**VPMOVSQD instruction (EVEX encoded version) reg-reg form**

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

FOR \(j := 0\) TO \(KL-1\)

\[i := j * 32\]
\[m := j * 64\]

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

\[\text{THEN } \text{DEST}[i+31:i] := \text{SaturateSignedQuadWordToDWord}(\text{SRC}[m+63:m])\]
\[\text{ELSE}\]

\[\text{IF } \text{merging-masking}\]
\[\text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged}\]
\[\text{ELSE } \text{DEST}[i+31:i] := 0\]

\[\text{FI}\]

\[\text{FI;}\]

ENDFOR

\[\text{DEST}[\text{MAXVL-1:VL/2}] := 0;\]
VPMOVQD instruction (EVEX encoded version) memory form

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VPMOVUSQD instruction (EVEX encoded version) reg-reg form

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI;
ENDFOR

DEST[MAXVL-1:VL/2] := 0;

VPMOVUSQD instruction (EVEX encoded version) memory form

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 32
    m := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged* ; merging-masking
        FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQD __m256i _mm512_cvtepi64_epi32( __m512i a);
VPMOVQD __m256i _mm512_mask_cvtepi64_epi32( __m512i a, __mmask8 k, __m512i a);
VPMOVQD void _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m256i _mm512_cvtusepi64_epi32( __m512i a);
VPMOVUSQD __m256i _mm512_mask_cvtusepi64_epi32( __m512i a, __mmask8 k, __m512i a);
VPMOVUSQD void _mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32( __m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32( __m256i a, __mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32( __m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32( __m128i a, __mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-53, "Type E6 Class Exception Conditions"; additionally:

#UD If EVEX.vvvv != 1111B.
VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

**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 (MAXVL-1:128/64/32) of the register destination are zeroed.

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

---

**Instruction Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Quarter Mem</td>
<td>ModRM.r/m (w)</td>
<td>ModRM.reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

---

**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.128.F3.0F38.W0 34 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed quad-word integers from xmm2 into 2 packed word integers in xmm1/m32 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 24 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m32 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 14 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned word integers in xmm1/m32 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 34 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed quad-word integers from ymm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 24 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 14 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 34 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed quad-word integers from zmm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 24 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 14 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.</td>
</tr>
</tbody>
</table>
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
ENDFOR
DEST[MAXVL-1:VL/4] := 0;

VPMOVQW instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

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] := 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
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

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

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] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+15:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;

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] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VPMOVSQW instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQW  __m128i  _mm512_cvtepi64_epi16( __m512i a);
VPMOVQW  __m128i  _mm512_mask_cvtepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVQW  __m128i  _mm512_maskz_cvtepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW  void  _mm512_mask_cvtepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i  _mm512_cvtepi64_epi16( __m512i a);
VPMOVSQW __m128i  _mm512_mask_cvtepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i  _mm512_maskz_cvtepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW  __m128i  _mm512_cvtepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVQW __m128i  _mm512_mask_cvtepi64_epi16( __m512i a);
VPMOVQW __m128i  _mm512_maskz_cvtepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW  __m128i  _mm512_cvtepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVQW __m128i  _mm512_mask_cvtepi64_epi16( __m512i a);
VPMOVQW __m128i  _mm512_maskz_cvtepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW  __m128i  _mm512_cvtepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVQW  void  _mm512_mask_cvtepi64_epi16( __m512i a);
VPMOVQW __m128i  _mm512_mask_cvtepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i  _mm512_maskz_cvtepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW  void  _mm512_mask_cvtepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Table 2-53, "Type E6 Class Exception Conditions"; additionally:

#UD  If EVEX.vvvv != 1111B.
VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

### Instruction 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.128.F3.0F38.W0 30 /r VPMOVWB xmm1/m64 {k1}[z], xmm2</td>
<td>A 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>A 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>A V/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Converts 8 packed unsigned word integers from xmm2 into 8 packed unsigned bytes in xmm1/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>A 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>A 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>A 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>A V/V</td>
<td>AVX512VL 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>A V/V</td>
<td>AVX512VL 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>A V/V</td>
<td>AVX512VL 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>Tuple Type</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>Half Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

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

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[MAXVL-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[MAXVL-1:VL/2] := 0;
VPMOVSWB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO 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[MAXVL-1:VL/2] := 0;

VPMOVUSWB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPMOVUSWB __m256i_mm512_cvtusepi16_epi8(__m512i a);</td>
<td>Down Convert Word to Byte</td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m256i_mm512_mask_cvtusepi16_epi8(__m256i a, __mmask32 k, __m512i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB void_mm512_mask_cvtusepi16_storeu_epi8(void *, __mmask32 k, __m512i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m256i_mm512_cvtsepi16_epi8(__m256i a);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m256i_mm512_mask_cvtsepi16_epi8(__m256i a, __mmask32 k, __m512i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB void_mm512_mask_cvtsepi16_storeu_epi8(void *, __mmask32 k, __m512i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m256i_mm512_cvtepi16_epi8(__m256i a);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m256i_mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB void_mm512_mask_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_cvtusepi16_epi8(__m256i a);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_mask_cvtusepi16_epi8(__m256i a, __mmask16 k, __m256i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB void_mm256_mask_cvtusepi16_storeu_epi8(void *, __mmask16 k, __m256i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_cvtsepi16_epi8(__m256i a);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_mask_cvtsepi16_epi8(__m256i a, __mmask16 k, __m256i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB void_mm256_mask_cvtsepi16_storeu_epi8(void *, __mmask16 k, __m256i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_cvtepi16_epi8(__m256i a);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_mask_cvtepi16_epi8(__m256i a, __mmask8 k, __m128i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB void_mm256_mask_cvtepi16_storeu_epi8(void *, __mmask8 k, __m128i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_cvtusepi16_epi8(__m256i a);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_mask_cvtusepi16_epi8(__m256i a, __mmask8 k, __m128i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB void_mm256_mask_cvtusepi16_storeu_epi8(void *, __mmask8 k, __m128i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_cvtsepi16_epi8(__m256i a);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_mask_cvtsepi16_epi8(__m256i a, __mmask8 k, __m128i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB void_mm256_mask_cvtsepi16_storeu_epi8(void *, __mmask8 k, __m128i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_cvtepi16_epi8(__m256i a);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB __m128i_mm256_mask_cvtepi16_epi8(__m256i a, __mmask8 k, __m128i b);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VPMOVUSWB void_mm256_mask_cvtepi16_storeu_epi8(void *, __mmask8 k, __m128i b);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Table 2-53, “Type E6 Class Exception Conditions”; additionally:

#UD If EVEX.vvvv != 1111B.
VPMULTISHIFTQB – Select Packed Unaligned Bytes from Quadword Sources

<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 83 /r VPMULTISHIFTQB xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI AVX512VL</td>
<td>Select unaligned bytes from qwords in xmm3/m128/m64bcst using control bytes in xmm2, write byte results to xmm1 under k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 83 /r VPMULTISHIFTQB ymm1 {k1}{z}, ymm2,ymm3/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI AVX512VL</td>
<td>Select unaligned bytes from qwords in ymm3/m256/m64bcst using control bytes in ymm2, write byte results to ymm1 under k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w1 83 /r VPMULTISHIFTQB zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI</td>
<td>Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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 selects eight unaligned bytes from each input qword element of the second source operand (the third operand) and writes eight assembled bytes for each qword element in the destination operand (the first operand). Each byte result is selected using a byte-granular shift control within the corresponding qword element of the first source operand (the second operand). Each byte result in the destination operand is updated under the writemask k1.

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

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

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

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

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

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

VPMULTISHIFTQB __m512i _mm512_multishift_epi64_epi8( __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_mask_multishift_epi64_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m512i _mm512_maskz_multishift_epi64_epi8( __mmask64 k, __m512i a, __m512i b);
VPMULTISHIFTQB __m256i _mm256_multishift_epi64_epi8( __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_mask_multishift_epi64_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m256i _mm256_maskz_multishift_epi64_epi8( __mmask32 k, __m256i a, __m256i b);
VPMULTISHIFTQB __m128i _mm_multishift_epi64_epi8( __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_mask_multishift_epi64_epi8(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULTISHIFTQB __m128i _mm_maskz_multishift_epi64_epi8( __mmask8 k, __m128i a, __m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-50, “Type E4NF Class Exception Conditions”. 
VPOPCNT — Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD

<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 54 /r</td>
<td>A V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 54 /r</td>
<td>A V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 54 /r</td>
<td>A V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 54 /r</td>
<td>A V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 54 /r</td>
<td>A V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 54 /r</td>
<td>A V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 55 /r</td>
<td>B V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 55 /r</td>
<td>B V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 55 /r</td>
<td>B V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 55 /r</td>
<td>B V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 55 /r</td>
<td>B V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 55 /r</td>
<td>B V/V</td>
<td>AVX512_VPOPCNTDQ AVX512VL</td>
<td>Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Full Mem</td>
<td>ModRMreg (w)</td>
<td>ModRMrmr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRMrmr/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

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

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

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

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

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

VPOPCNTW __m128i _mm_popcnt_epi16(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi16(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi16(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi16(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi16(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi16(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi16(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi16(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i _mm512_maskz_popcnt_epi16(__mmask32, __m512i);

VPOPCNTQ __m128i _mm_popcnt_epi64(__m128i);
VPOPCNTQ __m128i _mm_mask_popcnt_epi64(__m128i, __mmask8, __m128i);
VPOPCNTQ __m128i _mm_maskz_popcnt_epi64(__mmask8, __m128i);
VPOPCNTQ __m256i _mm256_popcnt_epi64(__m256i);
VPOPCNTQ __m256i _mm256_mask_popcnt_epi64(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i _mm256_maskz_popcnt_epi64(__mmask8, __m256i);
VPOPCNTQ __m512i _mm512_popcnt_epi64(__m512i);
VPOPCNTQ __m512i _mm512_mask_popcnt_epi64(__m512i, __mmask16, __m512i);
VPOPCNTQ __m512i _mm512_maskz_popcnt_epi64(__mmask16, __m512i);

VPOPCNTB __m128i _mm_popcnt_epi8(__m128i);
VPOPCNTB __m128i _mm_mask_popcnt_epi8(__m128i, __mmask16, __m128i);
VPOPCNTB __m128i _mm_maskz_popcnt_epi8(__mmask16, __m128i);
VPOPCNTB __m256i _mm256_popcnt_epi8(__m256i);
VPOPCNTB __m256i _mm256_mask_popcnt_epi8(__m256i, __mmask32, __m256i);
VPOPCNTB __m256i _mm256_maskz_popcnt_epi8(__mmask32, __m256i);
VPOPCNTB __m512i _mm512_popcnt_epi8(__m512i);
VPOPCNTB __m512i _mm512_mask_popcnt_epi8(__m512i, __mmask64, __m512i);
VPOPCNTB __m512i _mm512_maskz_popcnt_epi8(__mmask64, __m512i);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Type E4.
### VPROLD/VPROLVD/VPROLQ/VPROLVQ—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.128.66.0F38.W0 15 /r VPROLD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F0.W0 72 /1 ib VPROLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 15 /r VPROLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F0.W1 72 /1 ib VPROLVQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 15 /r VPROLD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F0.W0 72 /1 ib VPROLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 15 /r VPROLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F0.W1 72 /1 ib VPROLVQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 15 /r VPROLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F0.W0 72 /1 ib VPROLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 15 /r VPROLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F0.W1 72 /1 ib VPROLVQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</td>
<td>VEX.vvvv (w)</td>
<td>ModRM:r/m (R)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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 (MAXVL-1:128) of the corresponding ZMM register are zeroed.
EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.
EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC << COUNT) | (SRC >> (32 - COUNT));

LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC << COUNT) | (SRC >> (64 - COUNT));

VPROLD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[31:0], imm8)
    ELSE DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPROLVD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
            ELSE DEST[i+31:i] := LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31])
        FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+31:i] := 0
            FI;
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0

VPROLQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC1 *is memory*)
            THEN DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
            ELSE DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
        FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+63:i] := 0
            FI;
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0
VPROLQ (EVEX encoded versions)

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

FOR \( j \) := 0 TO \( KL-1 \)
\( i := j \times 64 \)
IF \( k1[j] \) OR *no writemask* THEN
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN \( \text{DEST}[i+63:i] := \text{LEFT\_ROTATE\_QWORDS}(\text{SRC1}[i+63:i], \text{SRC2}[63:0]) \)
    ELSE \( \text{DEST}[i+63:i] := \text{LEFT\_ROTATE\_QWORDS}(\text{SRC1}[i+63:i], \text{SRC2}[i+63:i]) \)
  FI;
ELSE
  IF *merging-masking*
    THEN *DEST*[i+63] remains unchanged*
  ELSE *zeroing-masking*
    \( \text{DEST}[i+63:i] := 0 \)
  FI
ENDFOR

\( \text{DEST}[\text{MAXVL}-1:VL] := 0 \)

Intel C/C++ Compiler Intrinsic Equivalent

\( \text{VPROLD} \_\_m512i \_\_mm512\_rol\_epi32(\_\_m512i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m512i \_\_mm512\_mask\_rol\_epi32(\_\_m512i \text{a}, \_\_mmask16 \text{k}, \_\_m512i \text{b}, \text{int imm}); \)
\( \text{VPROLD} \_\_m512i \_\_mm512\_maskz\_rol\_epi32(\_\_mmask16 \text{k}, \_\_m512i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m256i \_\_mm256\_rol\_epi32(\_\_m256i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m256i \_\_mm256\_mask\_rol\_epi32(\_\_m256i \text{a}, \_\_mmask8 \text{k}, \_\_m256i \text{b}, \text{int imm}); \)
\( \text{VPROLD} \_\_m256i \_\_mm256\_maskz\_rol\_epi32(\_\_mmask8 \text{k}, \_\_m256i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_rol\_epi32(\_\_m128i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_mask\_rol\_epi32(\_\_m128i \text{a}, \_\_mmask8 \text{k}, \_\_m128i \text{b}, \text{int imm}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_maskz\_rol\_epi32(\_\_mmask8 \text{k}, \_\_m128i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m512i \_\_mm\_rol\_epi64(\_\_m512i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m512i \_\_mm\_mask\_rol\_epi64(\_\_m512i \text{a}, \_\_mmask8 \text{k}, \_\_m512i \text{b}, \text{int imm}); \)
\( \text{VPROLD} \_\_m512i \_\_mm\_maskz\_rol\_epi64(\_\_mmask8 \text{k}, \_\_m512i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m256i \_\_mm\_rol\_epi64(\_\_m256i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m256i \_\_mm\_mask\_rol\_epi64(\_\_m256i \text{a}, \_\_mmask8 \text{k}, \_\_m256i \text{b}, \text{int imm}); \)
\( \text{VPROLD} \_\_m256i \_\_mm\_maskz\_rol\_epi64(\_\_mmask8 \text{k}, \_\_m256i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_rol\_epi64(\_\_m128i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_mask\_rol\_epi64(\_\_m128i \text{a}, \_\_mmask8 \text{k}, \_\_m128i \text{b}, \text{int imm}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_maskz\_rol\_epi64(\_\_mmask8 \text{k}, \_\_m128i \text{a}, \text{int imm}); \)
\( \text{VPROLD} \_\_m512i \_\_mm\_rol\_epi64(\_\_m512i \text{a}, \_\_mmask16 \text{k}, \_\_m512i \text{b}, \_\_m512i \text{cnt}); \)
\( \text{VPROLD} \_\_m512i \_\_mm\_mask\_rol\_epi64(\_\_m512i \text{a}, \_\_mmask16 \text{k}, \_\_m512i \text{b}, \_\_m512i \text{cnt}); \)
\( \text{VPROLD} \_\_m512i \_\_mm\_maskz\_rol\_epi64(\_\_mmask16 \text{k}, \_\_m512i \text{a}, \_\_m512i \text{cnt}); \)
\( \text{VPROLD} \_\_m256i \_\_mm\_rol\_epi64(\_\_m256i \text{a}, \_\_mmask16 \text{k}, \_\_m256i \text{b}, \_\_m256i \text{cnt}); \)
\( \text{VPROLD} \_\_m256i \_\_mm\_mask\_rol\_epi64(\_\_m256i \text{a}, \_\_mmask8 \text{k}, \_\_m256i \text{b}, \_\_m256i \text{cnt}); \)
\( \text{VPROLD} \_\_m256i \_\_mm\_maskz\_rol\_epi64(\_\_mmask8 \text{k}, \_\_m256i \text{a}, \_\_m256i \text{cnt}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_rol\_epi64(\_\_m128i \text{a}, \_\_mmask16 \text{k}, \_\_m128i \text{b}, \_\_m128i \text{cnt}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_mask\_rol\_epi64(\_\_m128i \text{a}, \_\_mmask8 \text{k}, \_\_m128i \text{b}, \_\_m128i \text{cnt}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_maskz\_rol\_epi64(\_\_mmask8 \text{k}, \_\_m128i \text{a}, \_\_m128i \text{cnt}); \)
\( \text{VPROLD} \_\_m512i \_\_mm\_rol\_epi64(\_\_m512i \text{a}, \_\_mmask16 \text{k}, \_\_m512i \text{b}, \_\_m512i \text{cnt}); \)
\( \text{VPROLD} \_\_m512i \_\_mm\_mask\_rol\_epi64(\_\_m512i \text{a}, \_\_mmask8 \text{k}, \_\_m512i \text{b}, \_\_m512i \text{cnt}); \)
\( \text{VPROLD} \_\_m512i \_\_mm\_maskz\_rol\_epi64(\_\_mmask8 \text{k}, \_\_m512i \text{a}, \_\_m512i \text{cnt}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_rol\_epi64(\_\_m128i \text{a}, \_\_mmask16 \text{k}, \_\_m128i \text{b}, \_\_m128i \text{cnt}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_mask\_rol\_epi64(\_\_m128i \text{a}, \_\_mmask8 \text{k}, \_\_m128i \text{b}, \_\_m128i \text{cnt}); \)
\( \text{VPROLD} \_\_m128i \_\_mm\_maskz\_rol\_epi64(\_\_mmask8 \text{k}, \_\_m128i \text{a}, \_\_m128i \text{cnt}); \)
SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions”. 

VPROLVQ __m128i _mm_mask_rolv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVQ __m128i _mm_maskz_rolv_epi64(__mmask8 k, __m128i a, __m128i cnt);
### VPRORD/VPRORVD/VPRORQ/VPRORVQ—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.128.66.0F38.W0 14 /r VPRORD xmm1 [k1]{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 14 /r VPRORQ xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 14 /r VPRORD ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 14 /r VPRORD zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 72 /0 ib VPRORD ymm1 [k1]{z}, ymm2/m256/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 72 /0 ib VPRORD zmm1 [k1]{z}, zmm2/m512/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 72 /0 ib VPRORQ zmm1 [k1]{z}, zmm2/m512/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</td>
<td>VEX.vvvv (w)</td>
<td>ModRM/r/m (R)</td>
<td>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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 (MAXVL-1:128) of the corresponding ZMM register are zeroed.

**EVEX.256 encoded version:** The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

**EVEX.512 encoded version:** The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

**Operation**

```plaintext
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC >> COUNT) | (SRC << (32 - COUNT));

RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC >> COUNT) | (SRC << (64 - COUNT));
```

**VPRORD (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS( SRC1[31:0], imm8)
      ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+31:i] := 0
      FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0
VPRORVD (EVEX encoded versions)

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

FOR \( j := 0 \) TO \( KL - 1 \)
  \( i := j \times 32 \)
  IF \( k1[j] \) OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN \( \text{DEST}[i+31:i] := \text{RIGHT\_ROTATE\_DWORDS}(\text{SRC1}[i+31:i], \text{SRC2}[31:0]) \)
    ELSE \( \text{DEST}[i+31:i] := \text{RIGHT\_ROTATE\_DWORDS}(\text{SRC1}[i+31:i], \text{SRC2}[i+31:i]) \)
  FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *\( \text{DEST}[i+31:i] \) remains unchanged* ; zeroing-masking
        \( \text{DEST}[i+31:i] := 0 \)
  FI
  FI;
ENDFOR
\( \text{DEST}[\text{MAXVL-1:VL}] := 0 \)

VPRORQ (EVEX encoded versions)

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

FOR \( j := 0 \) TO \( KL - 1 \)
  \( i := j \times 64 \)
  IF \( k1[j] \) OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN \( \text{DEST}[i+63:i] := \text{RIGHT\_ROTATE\_QWORDS}(\text{SRC1}[63:0], \text{imm8}) \)
    ELSE \( \text{DEST}[i+63:i] := \text{RIGHT\_ROTATE\_QWORDS}(\text{SRC1}[i+63:i], \text{imm8}) \)
  FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *\( \text{DEST}[i+63:i] \) remains unchanged* ; zeroing-masking
        \( \text{DEST}[i+63:i] := 0 \)
  FI
  FI
ENDFOR
\( \text{DEST}[\text{MAXVL-1:VL}] := 0 \)
VPRORVQ (EVEX encoded versions)

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

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

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPRORD __m512i _mm512_ror_epi32(__m512i a, int imm);
VPRORD __m512i _mm512_mask_ror_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPRORD __m512i _mm512_maskz_ror_epi32(__mmask16 k, __m512i a, int imm);
VPRORD __m256i _mm256_ror_epi32(__m256i a, int imm);
VPRORD __m256i _mm256_mask_ror_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORD __m256i _mm256_maskz_ror_epi32(__mmask8 k, __m256i a, int imm);
VPRORD __m128i _mm_ror_epi32(__m128i a, int imm);
VPRORD __m128i _mm_mask_ror_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORD __m128i _mm_maskz_ror_epi32(__mmask8 k, __m128i a, int imm);
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_epi32(__m128i a, __m128i b, __m128i cnt);
VPRORD __m128i _mm_mask_rorv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORD __m128i _mm_maskz_rorv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPRORD __m512i _mm512_rorv_epi64(__m512i a, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_mask_rorv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_maskz_rorv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPRORD __m512i _mm512_ror_epi32(__m512i a, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_mask_ror_epi32(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_maskz_ror_epi32(__mmask8 k, __m512i a, __m512i cnt);
VPRORD __m512i _mm512_rorv_epi64(__m512i a, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_mask_rorv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_maskz_rorv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPRORD __m512i _mm512_rorv_epi32(__m512i a, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_mask_rorv_epi32(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_maskz_rorv_epi32(__mmask8 k, __m512i a, __m512i cnt);
VPRORD __m512i _mm512_ror_epi64(__m512i a, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_mask_ror_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_maskz_ror_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPRORVQ __m128i _mm_mask_rorv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVQ __m128i _mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions".
VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, 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 A0 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter dword values to memory using writemask k1.</td>
</tr>
<tr>
<td>VPSCATTERDD vm32x {k1}, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A0 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter dword values to memory using writemask k1.</td>
</tr>
<tr>
<td>VPSCATTERDD vm32y {k1}, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A0 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter dword values to memory using writemask k1.</td>
</tr>
<tr>
<td>VPSCATTERDD vm32z {k1}, zmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A0 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter qword values to memory using writemask k1.</td>
</tr>
<tr>
<td>VPSCATTERDQ vm32x {k1}, xmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A0 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter qword values to memory using writemask k1.</td>
</tr>
<tr>
<td>VPSCATTERDQ vm32y {k1}, ymm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A0 /vsib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed dword indices, scatter qword values to memory using writemask k1.</td>
</tr>
<tr>
<td>VPSCATTERDQ vm32z {k1}, zmm1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

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 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
  \( i := j \times 32 \)
  \( k := j \times 64 \)
  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
  \( i := j \times 64 \)
  IF \( k1[j] \) OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEX[j+63:j]) * SCALE + DISP] := SRC[i+63:i]
    \( k1[j] := 0 \)
  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, __m256i 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, __m256i a, int scale);
```

```
VPSCATTERQD void _mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_i64scatter_epi32(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERQD void _mm_i64scatter_epi32(void * base, __m128i vdx, __m256i 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, __m256i 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, __m256i 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 Table 2-61, “Type E12 Class Exception Conditions”.
**VPSHLD — Concatenate and Shift Packed Data 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>EVEX.128.66.0F3A.W1 70 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 70 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 70 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 71 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 71 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 71 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 71 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 71 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 71 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

**Description**

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

This instruction supports memory fault suppression.
Operation

**VPSHLDW DEST, SRC2, SRC3, imm8**

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

FOR \(j := 0\) TO \(KL-1\):

IF \(\text{MaskBit}(j)\) OR *no writemask*:

\(\text{tmp} := \text{concat}(\text{SRC2}.\text{word}[j], \text{SRC3}.\text{word}[j]) \ll (\text{imm8} \& 15)\)

\(\text{DEST}.\text{word}[j] := \text{tmp}.\text{word}[1]\)

ELSE IF *zeroing*:

\(\text{DEST}.\text{word}[j] := 0\)

*ELSE DEST.word[j] remains unchanged*

\(\text{DEST}[\text{MAX}_V\text{L}-1:VL] := 0\)

**VPSHLDW DEST, SRC2, SRC3, imm8**

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

FOR \(j := 0\) TO \(KL-1\):

IF SRC3 is broadcast memop:

\(\text{tsrc3} := \text{SRC3}.\text{dword}[0]\)

ELSE:

\(\text{tsrc3} := \text{SRC3}.\text{dword}[j]\)

IF \(\text{MaskBit}(j)\) OR *no writemask*:

\(\text{tmp} := \text{concat}(\text{SRC2}.\text{dword}[j], \text{tsrc3}) \ll (\text{imm8} \& 31)\)

\(\text{DEST}.\text{dword}[j] := \text{tmp}.\text{dword}[1]\)

ELSE IF *zeroing*:

\(\text{DEST}.\text{dword}[j] := 0\)

*ELSE DEST.dword[j] remains unchanged*

\(\text{DEST}[\text{MAX}_V\text{L}-1:VL] := 0\)

**VPSHLDQ DEST, SRC2, SRC3, imm8**

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

FOR \(j := 0\) TO \(KL-1\):

IF SRC3 is broadcast memop:

\(\text{tsrc3} := \text{SRC3}.\text{qword}[0]\)

ELSE:

\(\text{tsrc3} := \text{SRC3}.\text{qword}[j]\)

IF \(\text{MaskBit}(j)\) OR *no writemask*:

\(\text{tmp} := \text{concat}(\text{SRC2}.\text{qword}[j], \text{tsrc3}) \ll (\text{imm8} \& 63)\)

\(\text{DEST}.\text{qword}[j] := \text{tmp}.\text{qword}[1]\)

ELSE IF *zeroing*:

\(\text{DEST}.\text{qword}[j] := 0\)

*ELSE DEST.qword[j] remains unchanged*

\(\text{DEST}[\text{MAX}_V\text{L}-1:VL] := 0\)
**Intel C/C++ Compiler Intrinsic Equivalent**

VPSHLD _m128i _mm_shldi_epi32(__m128i, __m128i, int);
VPSHLD _m128i _mm_mask_shldi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLD _m128i _mm_maskz_shldi_epi32(__mmask8, __m128i, __m128i, int);
VPSHLD _m256i _mm256_shldi_epi32(__m256i, __m256i, int);
VPSHLD _m256i _mm256_mask_shldi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLD _m256i _mm256_maskz_shldi_epi32(__mmask8, __m256i, __m256i, int);
VPSHLD _m512i _mm512_shldi_epi32(__m512i, __m512i, int);
VPSHLD _m512i _mm512_mask_shldi_epi32(__mmask16, __m512i, __m512i, int);
VPSHLD _m512i _mm512_maskz_shldi_epi32(__mmask16, __m512i, __m512i, int);
VPSHLDQ _m128i _mm_shldi_epi64(__m128i, __m128i, int);
VPSHLDQ _m128i _mm_mask_shldi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDQ _m128i _mm_maskz_shldi_epi64(__mmask8, __m128i, __m128i, int);
VPSHLDQ _m256i _mm256_shldi_epi64(__m256i, __m256i, int);
VPSHLDQ _m256i _mm256_mask_shldi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDQ _m256i _mm256_maskz_shldi_epi64(__mmask8, __m256i, __m256i, int);
VPSHLDQ _m512i _mm512_shldi_epi64(__m512i, __m512i, int);
VPSHLDQ _m512i _mm512_mask_shldi_epi64(__mmask32, __m512i, __m512i, int);
VPSHLDQ _m512i _mm512_maskz_shldi_epi64(__mmask32, __m512i, __m512i, int);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Type E4.
## VPSHLDV — Concatenate and Variable Shift Packed Data 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>EVEX.128.66.0F38.W1 70 /r VPSHLDVW xmm1{k1}{z}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 70 /r VPSHLDVW ymm1{k1}{z}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 70 /r VPSHLDVW zmm1{k1}{z}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the left by value in xmm3/m512 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 71 /r VPSHLDVD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 71 /r VPSHLDVD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 71 /r VPSHLDVD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the left by value in xmm3/m512 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 71 /r VPSHLDVQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 71 /r VPSHLDVQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 71 /r VPSHLDVQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the left by value in xmm3/m512 into zmm1.</td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Full Mem</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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

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

This instruction supports memory fault suppression.
**Operation**

FUNCTION concat(a, b):

IF words:
   d.word[1] := a
   d.word[0] := b
   return d
ELSE IF dwords:
   q.dword[1] := a
   q.dword[0] := b
   return q
ELSE IF qwords:
   o.qword[1] := a
   o.qword[0] := b
   return o

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

VPSHLDVD DEST, SRC2, SRC3
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1:
   IF SRC3 is broadcast memop:
      tsrc3 := SRC3.dword[0]
   ELSE:
      tsrc3 := SRC3.dword[j]
   IF MaskBit(j) OR *no writemask*:
      tmp := concat(DEST.dword[j], SRC2.dword[j]) << (tsrc3 & 31)
      DEST.dword[j] := tmp.dword[1]
   ELSE IF *zeroing*:
      DEST.dword[j] := 0
   *ELSE DEST.dword[j] remains unchanged*
DEST[MAX_VL-1:VL] := 0
VPSHLDVQ DEST, SRC2, SRC3
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1:
    IF SRC3 is broadcast memop:
        tsrc3 := SRC3.qword[0]
    ELSE:
        tsrc3 := SRC3.qword[j]
    IF MaskBit(j) OR *no writemask*:
        tmp := concat(DEST.qword[j], SRC2.qword[j]) << (tsrc3 & 63)
        DEST.qword[j] := tmp.qword[1]
    ELSE IF *zeroing*:
        DEST.qword[j] := 0
    *ELSE DEST.qword[j] remains unchanged*
    DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPSHLDVw __m128i __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVw __m128i __m128i _mm_mask_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVw __m128i __m128i _mm_maskz_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVw __m256i __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVw __m256i __m256i _mm256_mask_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVw __m256i __m256i _mm256_maskz_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVQ __m512i __m512i _mm512_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVQ __m512i __m512i _mm512_mask_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVQ __m512i __m512i _mm512_maskz_shldv_epi64(__m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Type E4.
VPSHRD — Concatenate and Shift Packed Data 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>EVEX.128.66.0F3A.W1 72 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 72 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 72 /r /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W0 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 73 /r /ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>imm8 (r)</td>
</tr>
</tbody>
</table>

**Description**

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

This instruction supports memory fault suppression.
Operation

**VPSHRDW DEST, SRC2, SRC3, imm8**

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

FOR j := 0 TO KL-1:
   IF MaskBit(j) OR *no writemask*:
      DEST.word[j] := concat(SRC3.word[j], SRC2.word[j]) >> (imm8 & 15)
   ELSE IF *zeroing*:
      DEST.word[j] := 0
   *ELSE DEST.word[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

**VPSHRDD DEST, SRC2, SRC3, imm8**

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

FOR j := 0 TO KL-1:
   IF SRC3 is broadcast memop:
      tsrc3 := SRC3.dword[0]
   ELSE:
      tsrc3 := SRC3.dword[j]
   IF MaskBit(j) OR *no writemask*:
      DEST.dword[j] := concat(tsrc3, SRC2.dword[j]) >> (imm8 & 31)
   ELSE IF *zeroing*:
      DEST.dword[j] := 0
   *ELSE DEST.dword[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

**VPSHRDQ DEST, SRC2, SRC3, imm8**

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

FOR j := 0 TO KL-1:
   IF SRC3 is broadcast memop:
      tsrc3 := SRC3.qword[0]
   ELSE:
      tsrc3 := SRC3.qword[j]
   IF MaskBit(j) OR *no writemask*:
      DEST.qword[j] := concat(tsrc3, SRC2.qword[j]) >> (imm8 & 63)
   ELSE IF *zeroing*:
      DEST.qword[j] := 0
   *ELSE DEST.qword[j] remains unchanged*

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

VPSHRDQ __m128i  _mm_shrdi_epi64(__m128i, __m128i, int);
VPSHRDQ __m128i  _mm_mask_shrdi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDQ __m128i  _mm_maskz_shrdi_epi64(__mmask8, __m128i, __m128i, int);
VPSHRDQ __m256i  _mm256_shrdi_epi64(__m256i, __m256i, int);
VPSHRDQ __m256i  _mm256_mask_shrdi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDQ __m256i  _mm256_maskz_shrdi_epi64(__mmask8, __m256i, __m256i, int);
VPSHRDQ __m512i  _mm512_shrdi_epi64(__m512i, __m512i, int);
VPSHRDQ __m512i  _mm512_mask_shrdi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHRDQ __m512i  _mm512_maskz_shrdi_epi64(__mmask8, __m512i, __m512i, int);

VPSHRDQ __m128i  _mm_shrdi_epi32(__m128i, __m128i, int);
VPSHRDQ __m128i  _mm_mask_shrdi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDQ __m128i  _mm_maskz_shrdi_epi32(__mmask8, __m128i, __m128i, int);
VPSHRDQ __m256i  _mm256_shrdi_epi32(__m256i, __m256i, int);
VPSHRDQ __m256i  _mm256_mask_shrdi_epi32(__m256i, __mmask16, __m256i, __m256i, int);
VPSHRDQ __m256i  _mm256_maskz_shrdi_epi32(__mmask16, __m256i, __m256i, int);
VPSHRDQ __m512i  _mm512_shrdi_epi32(__m512i, __m512i, int);
VPSHRDQ __m512i  _mm512_mask_shrdi_epi32(__m512i, __mmask32, __m512i, __m512i, int);
VPSHRDQ __m512i  _mm512_maskz_shrdi_epi32(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4.
VPSHRDV — Concatenate and Variable Shift Packed Data 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>EVEX.128.66.0F38.W1 72 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 72 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 72 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the right by value in xmm3/m512 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 73 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 73 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 73 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the right by value in xmm3/m512 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 73 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2 AVX512VL</td>
<td>Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 73 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VBMI2</td>
<td>Concatenate zmm1 and zmm2, extract result shifted to the right by value in xmm3/m512 into zmm1.</td>
</tr>
</tbody>
</table>

InstructionOperand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</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>Full Mem</td>
<td>ModRM:reg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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

Concatenate packed data, extract result shifted to the right by variable value.
This instruction supports memory fault suppression.
**Operation**

**VPSHRDVW DEST, SRC2, SRC3**

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

FOR j := 0 TO KL-1:

  IF MaskBit(j) OR *no writemask*:
    DEST.word[j] := concat(SRC2.word[j], DEST.word[j]) >> (SRC3.word[j] & 15)
  ELSE IF *zeroing*:
    DEST.word[j] := 0
  ELSE DEST.word[j] remains unchanged*

  DEST[MAX_VL-1:VL] := 0

**VPSHRDWDVW DEST, SRC2, SRC3**

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

FOR j := 0 TO KL-1:

  IF SRC3 is broadcast memop:
    tsrc3 := SRC3.dword[0]
  ELSE:
    tsrc3 := SRC3.dword[j]

  IF MaskBit(j) OR *no writemask*:
    DEST.dword[j] := concat(SRC2.dword[j], DEST.dword[j]) >> (tsrc3 & 31)
  ELSE IF *zeroing*:
    DEST.dword[j] := 0
  ELSE DEST.dword[j] remains unchanged*

  DEST[MAX_VL-1:VL] := 0

**VPSHRDQ DEST, SRC2, SRC3**

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

FOR j := 0 TO KL-1:

  IF SRC3 is broadcast memop:
    tsrc3 := SRC3.qword[0]
  ELSE:
    tsrc3 := SRC3.qword[j]

  IF MaskBit(j) OR *no writemask*:
    DEST.qword[j] := concat(SRC2.qword[j], DEST.qword[j]) >> (tsrc3 & 63)
  ELSE IF *zeroing*:
    DEST.qword[j] := 0
  ELSE DEST.qword[j] remains unchanged*

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

VPSHRDVQ __m128i __mm_shrdv_epi64(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i __mm_mask_shrdv_epi64(__m128i, __m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i __mm_maskz_shrdv_epi64(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i __mm256_shrdv_epi64(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i __mm256_mask_shrdv_epi64(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVQ __m256i __mm256_maskz_shrdv_epi64(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i __mm512_shrdv_epi64(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i __mm512_mask_shrdv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHRDVQ __m512i __mm512_maskz_shrdv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHRDVQ __m128i __mm256_shrdv_epi32(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i __mm256_mask_shrdv_epi32(__m128i, __mmask8, __m256i, __m128i, __m128i);
VPSHRDVQ __m128i __mm256_maskz_shrdv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m512i __mm512_shrdv_epi32(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i __mm512_mask_shrdv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHRDVQ __m512i __mm512_maskz_shrdv_epi32(__mmask16, __m512i, __m512i, __m512i);
VPSHRDVQ __m128i __mm256_shrdv_epi16(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i __mm256_mask_shrdv_epi16(__m128i, __mmask8, __m256i, __m128i, __m128i);
VPSHRDVQ __m128i __mm256_maskz_shrdv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m512i __mm512_shrdv_epi16(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i __mm512_mask_shrdv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHRDVQ __m512i __mm512_maskz_shrdv_epi16(__mmask32, __m512i, __m512i, __m512i);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Type E4.
VPSHUFBITQMB — Shuffle Bits from Quadword Elements Using Byte Indexes into Mask

<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 8F /r VPSHUFBITQMB k1{k2}, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Extract values in xmm2 using control bits of xmm3/m128 with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG AVX512VL</td>
<td>Extract values in ymm2 using control bits of ymm3/m256 with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 8F /r VPSHUFBITQMB k1{k2}, zmm2, zmm3/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG</td>
<td>Extract values in zmm2 using control bits of zmm3/m512 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>Tuple</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>Full Mem ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

Description

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

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

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFBITQMB __mmask16 __m512_bitshuffle_epi64_mask(__m128i, __m128i);
VPSHUFBITQMB __mmask16 __m512_bitshuffle_epi64_mask(__mmask16, __m128i, __m128i);
VPSHUFBITQMB __mmask32 __m256_bitshuffle_epi64_mask(__m256i, __m256i);
VPSHUFBITQMB __mmask32 __m256_bitshuffle_epi64_mask(__mmask32, __m256i, __m256i);
VPSHUFBITQMB __mmask64 __m512_bitshuffle_epi64_mask(__m512i, __m512i);
VPSHUFBITQMB __mmask64 __m512_bitshuffle_epi64_mask(__mmask64, __m512i, __m512i);
### 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.128.66.0F38.W0 47 /r VPSLLVD xmm1, xmm2, xmm3/m128</td>
<td>A</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.128.66.0F38.W1 47 /r VPSLLVQ xmm1, xmm2, xmm3/m128</td>
<td>A</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.256.66.0F38.W0 47 /r VPSLLVD ymm1, ymm2, ymm3/m256</td>
<td>A</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.256.66.0F38.W1 47 /r VPSLLVQ ymm1, ymm2, ymm3/m256</td>
<td>A</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.128.66.0F38.W1 12 /r VPSLLVW xmm1 {k1}{z}, xmm2, xmm3/m128</td>
<td>B</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.256.66.0F38.W1 12 /r VPSLLVW ymm1 {k1}{z}, ymm2, ymm3/m256</td>
<td>B</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.512.66.0F38.W1 12 /r VPSLLVW zmm1 {k1}{z}, zmm2, zmm3/m512</td>
<td>B</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.128.66.0F38.W0 47 /r VPSLLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>C</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.256.66.0F38.W0 47 /r VPSLLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>C</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.512.66.0F38.W0 47 /r VPSLLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>C</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.128.66.0F38.W1 47 /r VPSLLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</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.256.66.0F38.W1 47 /r VPSLLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</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.512.66.0F38.W1 47 /r VPSLLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</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>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvv (r)</td>
<td>ModRM:rm (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (r)</td>
<td>ModRM:rm (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (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 left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

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

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation
VPSLLVw (EVEX encoded version)
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
    ELSE
      IF *merging-masking*; merging-masking
        THEN *DEST[i+15:i] remains unchanged*;
        ELSE ; zeroing-masking
        DEST[i+15:i] := 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
VPSLLVD (VEX.128 version)
COUNT_0 := SRC2[31 : 0];
   (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 *)
COUNT_3 := SRC2[127 : 96];
IF COUNT_0 < 32 THEN
   DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
   DEST[31:0] := 0;
   (* Repeat shift operation for 2nd through 4th dwords *)
   IF COUNT_3 < 32 THEN
      DEST[127:96] := ZeroExtend(SRC1[127:96] << COUNT_3);
   ELSE
      DEST[127:96] := 0;
   DEST[MAXVL-1:128] := 0;
VPSLLVD (VEX.256 version)
COUNT_0 := SRC2[31 : 0];
   (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2 *)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
   DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
   DEST[31:0] := 0;
   (* Repeat shift operation for 2nd through 7th dwords *)
   IF COUNT_7 < 32 THEN
      DEST[255:224] := ZeroExtend(SRC1[255:224] << COUNT_7);
   ELSE
      DEST[255:224] := 0;
   DEST[MAXVL-1:256] := 0;
VPSLLVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
   i := j * 32
   IF k1[j] OR *no writemask* THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
         THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
         ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
      FI;
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+31:i] := 0
      FI
   FI
ENDFOR;
DEST[MAXVL-1:VL] := 0;
VPSLLVQ (VEX.128 version)

COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64 THEN
  DEST[63:0] := ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
  DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
  DEST[127:64] := ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
  DEST[127:96] := 0;
  DEST[MAXVL-1:128] := 0;

VPSLLVQ (VEX.256 version)

COUNT_0 := SRC2[63 : 0];
(*) Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 *)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64 THEN
  DEST[63:0] := ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
  DEST[63:0] := 0;
  (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
  DEST[255:192] := ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
  DEST[255:192] := 0;
  DEST[MAXVL-1:256] := 0;

VPSLLVQ (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
      ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
Intel C/C++ Compiler Intrinsic Equivalent

VPSLLW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m512i _mm512_sllv_epi32(__m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_mask_sllv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_maskz_sllv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m512i _mm512_sllv_epi64(__m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_mask_sllv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_maskz_sllv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSLLVQ __m256i _mm256_mask_sllv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVQ __m256i _mm256_maskz_sllv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVQ __m128i _mm_mask_sllv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m128i _mm_maskz_sllv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions”.
EVEX-encoded VPSLLVD/VPSLLVQ, see Table 2-49, “Type E4 Class Exception Conditions”.
EVEX-encoded VPSLLVW, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”.

INSTRUCTION SET REFERENCE, V-Z
## VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

<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>VEX.128.66.0F38.W0 46 /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 46 /r</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 11 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 11 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 11 /r</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 46 /r</td>
<td>C 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 46 /r</td>
<td>C 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 46 /r</td>
<td>C 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 46 /r</td>
<td>C 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 46 /r</td>
<td>C 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 46 /r</td>
<td>C 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>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</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 (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element is filled with the corresponding sign bit of the source element.

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

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation
VPSRAVW (EVEX encoded version)
\[(KL, VL) = (8, 128), (16, 256), (32, 512)\]

\[
\begin{align*}
\text{FOR } j := 0 \text{ TO } KL-1 \\
\quad i := j * 16 \\
\quad \text{IF } k1[j] \text{ OR } \text{no writemask}\n\quad \quad \text{THEN} \\
\quad \quad \quad \text{COUNT} := \text{SRC2}[i+3:i] \\
\quad \quad \quad \text{IF COUNT < 16} \\
\quad \quad \quad \quad \text{THEN } \text{DEST}[i+15:i] := \text{SignExtend}(	ext{SRC1}[i+15:i] >> \text{COUNT}) \\
\quad \quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \quad \text{FOR } k := 0 \text{ TO } 15 \\
\quad \quad \quad \quad \quad \text{DEST}[i+k] := \text{SRC1}[i+15] \\
\quad \quad \quad \quad \quad \text{ENDFOR;} \\
\quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \text{IF } \text{merging-masking} \\
\quad \quad \quad \quad \quad \text{THEN } \text{DEST}[i+15:i] \text{ remains unchanged} \\
\quad \quad \quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \quad \quad \text{DEST}[i+15:i] := 0 \\
\quad \quad \quad \text{FI} \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \quad \text{IF } \text{zeroing-masking} \\
\quad \quad \quad \quad \quad \text{THEN } \text{DEST}[i+15:i] := 0 \\
\quad \quad \quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \quad \text{DEST}[i+15:i] := 0 \\
\quad \quad \quad \text{FI} \\
\quad \text{FI} \\
\quad \text{ENDFOR;} \\
\quad \text{DEST}[\text{MAXVL-1:VL}] := 0;
\end{align*}
\]
VPSRAVD (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];
DEST[31:0] := SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] := SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] := 0;

VPSRAVD (VEX.256 version)
COUNT_0 := SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
DEST[31:0] := SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] := SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAXVL-1:256] := 0;

VPSRAVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
i := j * 32
IF k1[j] OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN
COUNT := SRC2[4:0]
IF COUNT < 32
THEN
DEST[i+31:i] := SignExtend(SRC1[i+31:i] >> COUNT)
ELSE
FOR k := 0 TO 31
DEST[i+k] := SRC1[i+31]
ENDFOR;
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;
ELSE
DEST[31:0] := 0
ENDFOR;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[31:0] remains unchanged* ; zeroing-masking
DEST[31:0] := 0
ELSE
DEST[31:0] := 0
ENDFOR;
DEST[MAXVL-1:VL] := 0;
VPSRAVQ (EVEX encoded version)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        COUNT := SRC2[5:0]
        IF COUNT < 64
          THEN  DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
          ELSE
            FOR k := 0 TO 63
              DEST[i+k] := SRC1[i+63]
            ENDFOR;
        FI
      ELSE
        COUNT := SRC2[i+5:i]
        IF COUNT < 64
          THEN  DEST[i+63:i] := SignExtend(SRC1[i+63:i] >> COUNT)
          ELSE
            FOR k := 0 TO 63
              DEST[i+k] := SRC1[i+63]
            ENDFOR;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN  *DEST[63:0] remains unchanged*
        ELSE ; zeroing-masking
          DEST[63:0] := 0
        FI
      FI
  ELSE ; merging-masking
    ENDFOR;
  DEST[MAXVL-1:VL] := 0;
**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRAVW __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVW __m256i _mm256_srav_epi32(__m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVW __m128i _mm_srav_epi32(__m128i a, __m128i cnt);
VPSRAVW __m128i _mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m512i _mm512_srav_epi64(__m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_mask_srav_epi64(__m512i s, __mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_maskz_srav_epi64(__mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m256i _mm256_srav_epi64(__m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_maskz_srav_epi64(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m128i _mm_srav_epi64(__m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_maskz_srav_epi64(__mmask8 m, __m128i a, __m128i cnt);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions”.
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions”.

VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic
### 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.128.66.0F38.W0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 45 /r VPSRLVQ xmm1, xmm2, xmm3/m128</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 45 /r VPSRLVQ ymm1, ymm2, ymm3/m256</td>
<td>A V/V</td>
<td>AVX2</td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 10 /r VPSRLVw xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 10 /r VPSRLVw ymm1 (k1)[z], ymm2, ymm3/m256</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 10 /r VPSRLVw zmm1 (k1)[z], zmm2, zmm3/m512</td>
<td>B 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 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 45 /r VPSRLVD xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>C 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 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 45 /r VPSRLVD ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>C 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 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 45 /r VPSRLVD zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>C 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 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 45 /r VPSRLVQ xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>C 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 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 45 /r VPSRLVQ ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>C 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 0s using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 45 /r VPSRLVQ zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>C 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 0s using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</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 right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

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

**VEX.256 encoded version:** The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

**EVEX encoded VPSRLVD/Q:** The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

**EVEX encoded VPSRLVW:** The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation**

**VPSRLVW (EVEX encoded version)**

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

\[
\begin{align*}
\text{FOR } j & := 0 \text{ TO } KL-1 \\
& i := j \times 16 \\
& \quad \text{IF } k1[j] \text{ OR *no writemask*} \\
& \quad \quad \text{THEN } \text{DEST}[i+15:i] := \text{ZeroExtend}(\text{SRC1}[i+15:i] \gg \text{SRC2}[i+15:i]) \\
& \quad \quad \text{ELSE} \\
& \quad \quad \quad \text{IF *merging-masking*} \\
& \quad \quad \quad \quad \text{THEN } \text{DEST}[i+15:i] \text{ remains unchanged} \\
& \quad \quad \quad \quad \text{ELSE} \\
& \quad \quad \quad \quad \quad \text{DEST}[i+15:i] := 0 \\
& \quad \quad \FI \\
& \FI \\
& \text{ENDFOR;} \quad \text{DEST}[\text{MAXVL-1:VL}] := 0; \\
\end{align*}
\]

**VPSRLVD (VEX.128 version)**

\[
\begin{align*}
& \text{COUNT}_0 := \text{SRC2}[31 : 0] \\
& \quad \text{(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)} \\
& \text{COUNT}_3 := \text{SRC2}[127 : 96]; \\
& \text{IF } \text{COUNT}_0 < 32 \text{ THEN} \\
& \quad \text{DEST}[31:0] := \text{ZeroExtend}(\text{SRC1}[31:0] \gg \text{COUNT}_0); \\
& \text{ELSE} \\
& \quad \text{DEST}[31:0] := 0; \\
& \quad \text{(* Repeat shift operation for 2nd through 4th dwords *)} \\
& \text{IF } \text{COUNT}_3 < 32 \text{ THEN} \\
& \quad \text{DEST}[127:96] := \text{ZeroExtend}(\text{SRC1}[127:96] \gg \text{COUNT}_3); \\
& \text{ELSE} \\
& \quad \text{DEST}[127:96] := 0; \\
& \text{DEST}[\text{MAXVL-1:128}] := 0; \\
\end{align*}
\]
VPSRLVD (VEX.256 version)
COUNT_0 := SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2 *)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
  DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
  DEST[31:0] := 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
  DEST[255:224] := ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
  DEST[255:224] := 0;
DEST[MAXVL-1:225] := 0;

VPSRLVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
    ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking*
      DEST[i+31:i] := 0
    FI
  FI
ENDFOR;
DEST[MAXVL-1:VL] := 0;

VPSRLVQ (VEX.128 version)
COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64 THEN
  DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
  DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
  DEST[127:64] := ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
  DEST[127:64] := 0;
DEST[MAXVL-1:128] := 0;
VPSRLVQ (VEX.256 version)
COUNT_0 := SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 *)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] := 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] := ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;

VPSRLVQ (EVEX encoded version)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*;
ELSE ; zeroing-masking
DEST[i+63:i] := 0
FI
ENDFOR;
DEST[MAXVL-1:VL] := 0;
Intel C/C++ Compiler Intrinsic Equivalent

VPSRLVW __m512i _mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_maskz_srlv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_srlv_epi32(__m256i m, __m256i count);
VPSRLVD __m512i _mm512_srlv_epi32(__m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_mask_srlv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_maskz_srlv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m256i _mm256_srlv_epi32(__m256i m, __m256i count);
VPSRLVQ __m512i _mm512_srlv_epi64(__m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_mask_srlv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_maskz_srlv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m256i _mm256_srlv_epi64(__m256i m, __m256i count);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Table 2-21, "Type 4 Class Exception Conditions".
EVEX-encoded VPSRLVD/Q, see Table 2-49, "Type E4 Class Exception Conditions".
EVEX-encoded VPSRLVW, see Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions".
## VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

<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 25 / r ib</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512VL</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>VPTERNLOGD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 25 / r ib</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512VL</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>VPTERNLOGD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 25 / r ib</td>
<td>A/V</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>VPTERNLOGD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 25 / r ib</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512VL</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>VPTERNLOGQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 25 / r ib</td>
<td>A/V</td>
<td>V/V</td>
<td>AVX512VL</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>VPTERNLOGQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 25 / r ib</td>
<td>A/V</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>
<tr>
<td>VPTERNLOGQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/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 5-9 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>
<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

\( i := 32 \times j \)

IF k1[j] OR *no writemask*

THEN

FOR k := 0 TO 31

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

THEN DEST[i+k] := imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]

ELSE DEST[i+k] := imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]

FI;

; table lookup of immediate bellow;

ELSE

IF *merging-masking* ; merging-masking

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

ELSE ; zeroing-masking

DEST[31+i+i] := 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] := 0
VPTERNLOGQ (EVEX encoded versions)

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

\[
\text{FOR } j := 0 \text{ TO } KL-1
\]
\[
i := j \times 64
\]
\[
\text{IF } k1[j] \text{ OR } \neg \text{writemask} \text{ THEN}
\]
\[
\text{FOR } k := 0 \text{ TO } 63
\]
\[
\text{IF (EVEX.b = 1) AND (SRC2 *is memory*)}
\]
\[
\text{THEN } \text{DEST}[j][k] := \text{imm}[\text{DEST}[i+k] \ll 2] + (\text{SRC1}[i+k] \ll 1) + \text{SRC2}[k]
\]
\[
\text{ELSE } \text{DEST}[j][k] := \text{imm}[\text{DEST}[i+k] \ll 2] + (\text{SRC1}[i+k] \ll 1) + \text{SRC2}[i+k]
\]
\[
F;
\]
\[
\text{ELSE}
\]
\[
\text{IF } \neg \text{merging-mask}
\]
\[
\text{THEN } \text{DEST}[63+i:i] \text{ remains unchanged}
\]
\[
\text{ELSE}
\]
\[
\text{zeroing-mask}
\]
\[
\text{DEST}[63+i] := 0
\]
\[
F;
\]
\[
\text{ENDFOR};
\]
\[
\text{DEST[MAXVL-1:VL]} := 0
\]

**Intel C/C++ Compiler Intrinsic Equivalents**

\[
\text{VPTERNLOGD } \text{__m512i } \text{__mm512_ternarylogic_epi32}(\text{__m512i } a, \text{__m512i } b, \text{int } \text{imm})
\]
\[
\text{VPTERNLOGD } \text{__m512i } \text{__mm512_mask_ternarylogic_epi32}(\text{__m512i } s, \text{__mmask16 } m, \text{__m512i } a, \text{__m512i } b, \text{int } \text{imm})
\]
\[
\text{VPTERNLOGD } \text{__m512i } \text{__mm512_maskz_ternarylogic_epi32}(\text{__mmask } m, \text{__m512i } a, \text{__m512i } b, \text{int } \text{imm})
\]
\[
\text{VPTERNLOGD } \text{__m256i } \text{__mm256_ternarylogic_epi32}(\text{__m256i } a, \text{__m256i } b, \text{int } \text{imm})
\]
\[
\text{VPTERNLOGD } \text{__m256i } \text{__mm256_mask_ternarylogic_epi32}(\text{__mmask8 } m, \text{__m256i } a, \text{__m256i } b, \text{int } \text{imm})
\]
\[
\text{VPTERNLOGD } \text{__m256i } \text{__mm256_maskz_ternarylogic_epi32}(\text{__mmask8 } m, \text{__m256i } a, \text{__m256i } b, \text{int } \text{imm})
\]
\[
\text{VPTERNLOGD } \text{__m128i } \text{__mm128_ternarylogic_epi32}(\text{__m128i } a, \text{__m128i } b, \text{int } \text{imm})
\]
\[
\text{VPTERNLOGD } \text{__m128i } \text{__mm128_mask_ternarylogic_epi32}(\text{__mmask8 } m, \text{__m128i } a, \text{__m128i } b, \text{int } \text{imm})
\]
\[
\text{VPTERNLOGD } \text{__m128i } \text{__mm128_maskz_ternarylogic_epi32}(\text{__mmask8 } m, \text{__m128i } a, \text{__m128i } b, \text{int } \text{imm})
\]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions".
### VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set 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.128.66.0F38.W0 26 /r VPTESTMB k2 {k1}, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>AVX512VL</td>
<td>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.256.66.0F38.W0 26 /r VPTESTMB k2 {k1}, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>AVX512VL</td>
<td>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.512.66.0F38.W0 26 /r VPTESTMB k2 {k1}, zmm2, zmm3/m512</td>
<td>A/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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 26 /r VPTESTMW k2 {k1}, xmm2, xmm3/m128</td>
<td>A/V/V</td>
<td>AVX512VL</td>
<td>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.256.66.0F38.W1 26 /r VPTESTMW k2 {k1}, ymm2, ymm3/m256</td>
<td>A/V/V</td>
<td>AVX512VL</td>
<td>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.512.66.0F38.W1 26 /r VPTESTMW k2 {k1}, zmm2, zmm3/m512</td>
<td>A/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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 27 /r VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst</td>
<td>B/V/V</td>
<td>AVX512VL</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.256.66.0F38.W0 27 /r VPTESTMD k2 {k1}, ymm2, ymm3/m256/m32bcst</td>
<td>B/V/V</td>
<td>AVX512VL</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.512.66.0F38.W0 27 /r VPTESTMD k2 {k1}, zmm2, zmm3/m512/m32bcst</td>
<td>B/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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, xmm2, xmm3/m128/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL</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.256.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, ymm2, ymm3/m256/m64bcst</td>
<td>B/V/V</td>
<td>AVX512VL</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.512.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, zmm2, zmm3/m512/m64bcst</td>
<td>B/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>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full Mem</td>
<td>ModRM/reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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 bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

VPTESTMD/VPTESTMQ: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

VPTESTMB/VPTESTMW: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

Operation

VPTESTMB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j := 0 TO KL-1
  i := j * 8
  IF k1[j] OR *no writemask*
    THEN DEST[j] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;
    ELSE DEST[j] = 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPTESTMW (EVEX encoded versions)

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;
    ELSE DEST[j] = 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0

VPTESTMD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;
        ELSE DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;
      FI;
    ELSE DEST[j] := 0 ; zeroing-masking only
  FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
VPTESTMQ (EVEX encoded versions)

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

\[
\text{FOR } j := 0 \text{ TO } KL-1
\]

\[
i := j * 64
\]

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

\[
\begin{align*}
\text{THEN} \\
\text{IF } (EVEX.b = 1) \text{ AND (SRC2 *is memory*)} \\
\text{THEN } \text{DEST}[j] := (SRC1[i+63:i] \text{ BITWISE AND SRC2}[63:0] \neq 0)? 1 : 0; \\
\text{ELSE } \text{DEST}[j] := (SRC1[i+63:i] \text{ BITWISE AND SRC2}[i+63:i] \neq 0)? 1 : 0; \\
\end{align*}
\]

\[
\text{ELSE } \text{DEST}[j] := 0 ; \text{zeroing-masking only}
\]

\[
\text{ENDFOR}
\]

\[
\text{DEST[MAX}_KL-1:KL] := 0
\]

**Intel C/C++ Compiler Intrinsic Equivalents**

\[
\begin{align*}
\text{VPTESTMB } & \_\text{mmask64 } \_\text{mm512\_test\_epi8\_mask( } \_\text{m512i a, } \_\text{m512i b);} \\
\text{VPTESTMB } & \_\text{mmask64 } \_\text{mm512\_mask\_test\_epi8\_mask( } \_\text{mmask64, } \_\text{m512i a, } \_\text{m512i b);} \\
\text{VPTESTMw } & \_\text{mmask32 } \_\text{mm512\_test\_epi16\_mask( } \_\text{m512i a, } \_\text{m512i b);} \\
\text{VPTESTMw } & \_\text{mmask32 } \_\text{mm512\_mask\_test\_epi16\_mask( } \_\text{mmask32, } \_\text{m512i a, } \_\text{m512i b);} \\
\text{VPTESTMD } & \_\text{mmask16 } \_\text{mm512\_test\_epi32\_mask( } \_\text{m512i a, } \_\text{m512i b);} \\
\text{VPTESTMD } & \_\text{mmask16 } \_\text{mm512\_mask\_test\_epi32\_mask( } \_\text{mmask16, } \_\text{m512i a, } \_\text{m512i b);} \\
\text{VPTESTMQ } & \_\text{mmask8 } \_\text{mm512\_test\_epi64\_mask( } \_\text{m512i a, } \_\text{m512i b);} \\
\text{VPTESTMQ } & \_\text{mmask8 } \_\text{mm512\_mask\_test\_epi64\_mask( } \_\text{mmask8, } \_\text{m512i a, } \_\text{m512i b);}
\end{align*}
\]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VPTESTMD/Q: See Table 2-49, “Type E4 Class Exception Conditions”.

VPTESTMB/W: See Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”.
### 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.128.F3.0F38.W0 26 /r VPTESTNMB k2 [k1], xmm2, xmm3/m128</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 26 /r VPTESTNMB k2 [k1], ymm2, ymm3/m256</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 26 /r VPTESTNMB k2 [k1], zmm2, zmm3/m512</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 26 /r VPTESTNMw k2 [k1], xmm2, xmm3/m128</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 26 /r VPTESTNMw k2 [k1], ymm2, ymm3/m256</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 26 /r VPTESTNMw k2 [k1], zmm2, zmm3/m512</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 27 /r VPTESTNMD k2 [k1], xmm2, xmm3/m128/m32bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 27 /r VPTESTNMD k2 [k1], ymm2, ymm3/m256/m32bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 27 /r VPTESTNMD k2 [k1], zmm2, zmm3/m512/m32bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 27 /r VPTESTNMQ k2 [k1], xmm2, xmm3/m128/m64bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 27 /r VPTESTNMQ k2 [k1], ymm2, ymm3/m256/m64bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 27 /r VPTESTNMQ k2 [k1], zmm2, zmm3/m512/m64bcst</td>
<td>B 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>
<td></td>
</tr>
</tbody>
</table>
Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full Mem</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</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 bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

**Operation**

**VPTESTNMB**

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

FOR j := 0 TO KL-1
  i := j*8
  IF MaskBit(j) OR *no writemask*
    THEN
      DEST[j] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0
    ELSE DEST[j] := 0; zeroing masking only
  FI
ENDFOR
DEST[MAX_KL-1:KL] := 0

**VPTESTNMW**

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

FOR j := 0 TO KL-1
  i := j*16
  IF MaskBit(j) OR *no writemask*
    THEN
      DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0
    ELSE DEST[j] := 0; zeroing masking only
  FI
ENDFOR
DEST[MAX_KL-1:KL] := 0
VPTESTNMD

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

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

DEST[MAX_KL-1:KL] := 0

VPTESTNMQ

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

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

DEST[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPTESTNMB __mmask64 __mm512_testn_epi8_mask( __m512i a, __m512i b);
VPTESTNMB __mmask64 __mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 __mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTESTNMB __mmask32 __mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 __mm_testn_epi8_mask(__m128i a, __m128i b);
VPTESTNMB __mmask16 __mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMW __mmask32 __mm512_testn_epi16_mask( __m512i a, __m512i b);
VPTESTNMW __mmask32 __mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTNMW __mmask16 __mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTESTNMW __mmask16 __mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTESTNMW __mmask8 __mm_testn_epi16_mask(__m128i a, __m128i b);
VPTESTNMW __mmask8 __mm_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);

VPTESTNMD __mmask16 __mm512_testn_epi32_mask( __m512i a, __m512i b);
VPTESTNMD __mmask16 __mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMD __mmask8 __mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTESTNMD __mmask8 __mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMD __mmask8 __mm_testn_epi32_mask(__m128i a, __m128i b);
VPTESTNMD __mmask8 __mm_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);

VPTESTNMQ __mmask8 __mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTESTNMQ __mmask8 __mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMQ __mmask8 __mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTESTNMQ __mmask8 __mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMQ __mmask8 __mm_testn_epi64_mask(__m128i a, __m128i b);

VPTESTNMB/W/D/Q—Logical NAND and Set
VPTESTNMQ __mmask8 __mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VPTESTNMD/VPTESTNMQ: See Table 2-49, "Type E4 Class Exception Conditions".

VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions".
VRANGEPD—Range Restriction Calculation For Packed Pairs of 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 50 / r ib VRANGEPD xmm1 [k1]{z}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>A</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.256.66.0F3A.W1 50 / r ib VRANGEPD ymm1 [k1]{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A</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.512.66.0F3A.W1 50 / r ib VRANGEPD zmm1 [k1]{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8</td>
<td>A</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>Tuple Type</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>Full</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**

VRANGEPD—Range Restriction Calculation For Packed Pairs of Float64 Values
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 5-10. 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 5-10.

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 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-12.

<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>
**Operation**

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])

{
    // Check if SNAN and report IE, see also Table 5-10
    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 5-11
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 5-12
    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];
VRANGEPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
      ELSE DEST[i+63:i] := RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] = 0
      FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;

Where:
  zmm_dst is the destination operand.
  zmm_src is the input operand to compare against ±1023 (this is SRC1).
  zmm_1023 is the reference operand, contains the value of 1023 (and this is SRC2).
  IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.

In case |zmm_src| < 1023 (i.e., SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 1023 (received on zmm_1023, which is SRC2).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 1023, the selected sign of SRC1 is kept.

Thus, if zmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023 while if zmm_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 ( __m512 d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_maskz_range_pd ( __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_maskz_range_round_pd ( __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d _mm256_range_pd ( __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_mask_range_pd ( __m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_maskz_range_pd ( __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d _mm_range_pd ( __m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm_mask_range_pd ( __m128 d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm_maskz_range_pd ( __mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions”. 
VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Op/En Type</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>Full</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/m (r)</td>
<td>Imm8</td>
<td></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.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.

- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-10. 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 5-10.

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 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-12.
Operation
RangeSP(SCR1[31:0], SCR2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-10
    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 5-11
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-12
    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];
The following example describes a common usage of this instruction for checking that the input operand is bound-
ed 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 Table 2-46, "Type E2 Class Exception Conditions".
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.LLIG.66.0F3A.W1 S1 /r VRANGESD xmm1 {k1}[z], xmm2, xmm3/m64{sae}, imm8</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</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.

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 128:63 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 5-10. 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 5-10.

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 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-12.
Operation

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-10
    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 5-11 ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 5-12 ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
    ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[63] << 63) OR (TMP[62:0]); // Preserve Src1 sign bit
    01: dest := TMP[63:0]; // Preserve sign of compare result
    10: dest := (0 << 63) OR (TMP[62:0]); // Zero out sign bit
    11: dest := (1 << 63) OR (TMP[62:0]); // Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0] = imm8[1:0];
SignSelCtl[1:0] = imm8[3:2];
VRANGESD
IF k1[0] OR *no writemask*
  THEN DEST[63:0] := RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[63:0] remains unchanged*
  ELSE ; zeroing-masking
    DEST[63:0] = 0
  FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±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 VRANGEPD will be the minimal value of -1023 while 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 Table 2-47, “Type E3 Class Exception Conditions”.
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.LLIG.66.0F3A.W0 S1 /r</td>
<td>A</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>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</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 5-10. 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 5-10.

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 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-12.
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 5-10
   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 5-11
   ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-12
   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[127:32].:= SRC1[127:32]
DEST[MAXVL-1:128].:= 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±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 Table 2-47, "Type E3 Class Exception Conditions".
VRCP14PD—Compute Approximate Reciprocals 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 4C /r</td>
<td>A</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>A</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>A</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>Tuple Type</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>Full</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 ∞ 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 5-13. VRCP14PD/VRCP14SD Special Cases**

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ≤ X ≤ 2^{-1024}</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>-2^{-1024} ≤ X ≤ 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 ((EVEX encoded versions))**

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

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+63:i] := APPROXIMATE(1.0/SRC[63:0]);
    ELSE DEST[i+63:i] := APPROXIMATE(1.0/SRC[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

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

```
VRCP14PD __m512d _mm512_rcp14_pd( __m512d a);
VRCP14PD __m512d _mm512_mask_rcp14_pd(__m512d s, __mmask8 k, __m512d a);
VRCP14PD __m512d _mm512_maskz_rcp14_pd(__mmask8 k, __m512d a);
```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions".
VRCP14SD—Compute Approximate Reciprocal 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.LLIG.66.0F38.W1 4D /r</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvv (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 $\infty$ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-13 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at:


**Operation**

**VRCP14SD (EVEX version)**

IF k1[0] OR "no writemask"
  THEN DEST[63:0] := APPROXIMATE(1.0/SRC2[63:0]);
ELSE
  IF "merging-masking" ; merging-masking
    THEN "DEST[63:0] remains unchanged"
  ELSE ; zeroing-masking
    DEST[63:0] := 0
  FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SD __m128d _mm_rcp14_sd(__m128d a, __m128d b);
VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d _mm_maskz_rcp14_sd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-51, "Type E5 Class Exception Conditions".
VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

Opcode/Instruction | Op/en | 64/32 bit Mode Support | CPUID Feature Flag | Description
---|---|---|---|---
EVEX.128.66.0F38.W0 4C/r VRCP14PS xmm1 {k1}{z}, xmm2/m128/m32bcst | A | V/V | AVX512VL AVX512F | Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4C/r VRCP14PS ymm1 {k1}{z}, ymm2/m256/m32bcst | A | V/V | AVX512VL AVX512F | Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4C/r VRCP14PS zmm1 {k1}{z}, zmm2/m512/m32bcst | A | V/V | AVX512F | Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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 $\infty$ 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 5-14. VRCP14PS/VRCP14SS Special Cases

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \leq X \leq 2^{-128}$</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$-2^{-128} \leq X \leq -0$</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[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PS __m512 __m512_rcp14_ps(__m512 a);
VRCP14PS __m512 __m512_mask_rcp14_ps(__m512 s, __mmask16 k, __m512 a);
VRCP14PS __m512 __m512_maskz_rcp14_ps(__mmask16 k, __m512 a);
VRCP14PS __m256 __m256_rcp14_ps(__m256 a);
VRCP14PS __m256 __m256_mask_rcp14_ps(__m256 s, __mmask8 k, __m256 a);
VRCP14PS __m256 __m256_maskz_rcp14_ps(__mmask8 k, __m256 a);
VRCP14PS __m128 __m128_rcp14_ps(__m128 a);
VRCP14PS __m128 __m128_mask_rcp14_ps(__m128 s, __mmask8 k, __m128 a);
VRCP14PS __m128 __m128_maskz_rcp14_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-49, "Type E4 Class Exception Conditions".
**VRCP14SS—Compute Approximate Reciprocal 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.LLG.66.0F38.W0 4D /r VRCP14SS xmm1 {k1}[z], xmm2, xmm3/m32</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>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 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.</td>
<td></td>
</tr>
</tbody>
</table>

**Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRM:reg (w)</td>
<td>Evm.evvv (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 $\infty$ 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 will be 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 5-14 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;

FI;

DEST[MAXVL-1:128] := 0
Intel 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 Table 2-51, "Type E5 Class Exception Conditions".
VREDECPD—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 ib VREDECPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed double-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 56 /r ib VREDECPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Perform reduction transformation on packed double-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 56 /r ib VREDECPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Perform reduction transformation on double-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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 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.

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

![Figure 5-28. Imm8 Controls for VREDECPD/SD/PS/SS](image-url)
Handling of special case of input values are listed in Table 5-15.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Round Mode</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>Src_1</td>
<td>&lt; 2^{-M-1}$</td>
</tr>
<tr>
<td>$</td>
<td>Src_1</td>
<td>&lt; 2^{-M}$</td>
</tr>
<tr>
<td></td>
<td>RPI, $Src_1 \leq 0$</td>
<td>$Src_1$</td>
</tr>
<tr>
<td></td>
<td>RNI, $Src_1 &gt; 0$</td>
<td>$Src_1$</td>
</tr>
<tr>
<td></td>
<td>RNI, $Src_1 &lt; 0$</td>
<td>Round ($Src_1+2^{M}$) *</td>
</tr>
<tr>
<td>$Src_1 = \pm 0$, or Dest $= \pm 0$ ($Src_1!=INF$)</td>
<td>NOT RNI</td>
<td>+0.0</td>
</tr>
<tr>
<td>$Src_1 = \pm INF$</td>
<td>any</td>
<td>+0.0</td>
</tr>
<tr>
<td>$Src_1= \pm NAN$</td>
<td>n/a</td>
<td>QNaN($Src_1$)</td>
</tr>
</tbody>
</table>

* Round control $= (imm8.MS1)\? MXCSR.RC: imm8.RC

**Operation**

ReduceArgumentDP($SRC[63:0]$, imm8[7:0])

```c
// 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 := imm[3]; // 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 $KL-1$

```c
i := j * 64
IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC *is memory*)
        THEN DEST[i+63:i] := ReduceArgumentDP($SRC[63:0]$, imm8[7:0]);
        ELSE DEST[i+63:i] := ReduceArgumentDP($SRC[i+63:i]$, imm8[7:0]);
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged* ; merging-masking
            ELSE ; zeroing-masking
                DEST[i+63:i] = 0
            FI;
        FI;
ENDIF;
ENDFOR;
DEST[MAXVL-1:VL] := 0
```

**VREDUCEPD**—Perform Reduction Transformation on Packed Float64 Values
**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCEPD __m512d _mm512_mask_reduce_pd( __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_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd( __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 Table 2-46, "Type E2 Class Exception Conditions"; 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.LLIG.66.0F3A.W1 57 VREDUCESD xmm1 [k1][z], xmm2, xmm3/m64{sa}, imm8/r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</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>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRM:reg (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 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:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M \cdot \text{src})) \cdot 2^{-M};
\]

where "\text{Round()}" treats "\text{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

- If RC = RNE: \( 0 \leq |\text{Reduced Result}| \leq 2^p - 1 \)
- If RC \neq RNE: \( 0 \leq |\text{Reduced Result}| < 2^p \)

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 5-15.

**Operation**

ReduceArgumentDP([SRC[63:0], imm8[7:0]])

```plaintext
{
    // 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 := imm[3]; // Suppress Precision Exception
    TMP[63:0] := 2^M \cdot \text{ROUND}(2^M \cdot \text{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
}
```
VREDUCESD
IF k1[0] or "no writemask"
    THEN DEST[63:0] := ReduceArgumentDP(SRC2[63:0], imm8[7:0])
ELSE
    IF "merging-masking"
        THEN "DEST[63:0] remains unchanged"
    ELSE
        THEN DEST[63:0] = 0
    FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

SIMD Floating-Point Exceptions
Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Table 2-47, "Type E3 Class Exception Conditions".
VREDUCEPS—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 VREDUCEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8</td>
<td>A</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 VREDUCEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8</td>
<td>A</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 VREDUCEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{saе}, imm8</td>
<td>A</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>Tuple Type</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>Full</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 an unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M*\text{src}))*2^{-M};
\]

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 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.

Handling of special case of input values are listed in Table 5-15.
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 := imm[3]; // 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
}

VREDUCEPS
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
i := j * 32
IF k1[j] OR *no writemask* THEN
IF (EVEX.b == 1) AND (SRC *is memory*)
THEN DEST[i+31:i] := ReduceArgumentSP(SRC[31:0], imm8[7:0]);
ELSE DEST[i+31:i] := ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+31:i] = 0
FI;
FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPS __m512 _mm512_mask_reduce_ps( __m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_mask_reduce_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_maskz_reduce_ps(__mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m256 _mm256_mask_reduce_ps( __m256 a, int imm)
VREDUCEPS __m256 _mm256_mask_reduce_ps(__m256 s, __mmask8 k, __m256 a, int imm)
VREDUCEPS __m256 _mm256_maskz_reduce_ps(__mmask8 k, __m256 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps( __m128 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps(__m128 s, __mmask8 k, __m128 a, int imm)
VREDUCEPS __m128 _mm_maskz_reduce_ps(__mmask8 k, __m128 a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Table 2-46, "Type E2 Class Exception Conditions"; additionally:
#UD IF EVEX.vvvv != 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.LLIG.66.0F3A.W0 57 /r</td>
<td>A</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>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</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 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 an unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src - (ROUND(2M * src)) * 2^-M;

where "Round()" treats "src", "2M", 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.

Handling of special case of input values are listed in Table 5-15.

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 := imm[3]; // 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[MAXVL-1:128] := 0

---

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

VREDUCESS __m128 _mm_mask_reduce_ss( __m128 a, __m128 b, int imm, int sae)

VREDUCESS __m128 _mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)

VREDUCESS __m128 _mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)

---

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Table 2-47, "Type E3 Class Exception Conditions".
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</td>
<td>A</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</td>
<td>A</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</td>
<td>A</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>Tuple Type</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>Full</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

\[
\text{ROUND}(x) = 2^{-\text{M}} \times \text{Round}_{\text{to INT}}(x \times 2^{\text{M}}, \text{round_ctrl}),
\]
\[
\text{round_ctrl} = \text{imm}[3:0];
\]
\[
\text{M} = \text{imm}[7:4];
\]
The operation of \(x \times 2^{\text{M}}\) is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).
VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),
round_ctrl = imm[3:0];

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

Table 5-16. VRNDSCALEPD/SD/PS/SS Special Cases

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Src1=±inf</th>
<th>Src1=±NAN</th>
<th>Src1=±0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td></td>
<td>Src1 converted to QNAN</td>
<td></td>
</tr>
</tbody>
</table>
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 := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := RoundToIntegerDP((TMP_SRC[i+63:i], imm8[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPD __m512d _mm512_roundscale_pd( __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_roundscale_round_pd( __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_mask_roundscale_pd( __m512d s, __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_mask_roundscale_round_pd( __m512d s, __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_pd( __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_round_pd( __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_roundscale_pd( __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_mask_roundscale_pd( __m256d s, __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_pd( __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m128d _mm_roundscale_pd( __m128d a, int imm);
VRNDSCALEPD __m128d _mm_mask_roundscale_pd( __m128d s, __mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm_maskz_roundscale_pd( __mmask8 k, __m128d a, int imm);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions".
VRNDSCALESĐ—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.LLIG.66.0F3A.W1 0B /r ib VRNDSCALESĐ xmm1 {k1}{z}, xmm2, xmm3/m64{sa}, imm8</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM/r/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 of 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 first operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESĐ 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).

VRNDSCALESĐ 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 5-16.
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[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VRNDSCALESD __m128d _mm_roundscale_sd ( __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_roundscale_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_mask_roundscale_sd ( __m128d s, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_mask_roundscale_round_sd ( __m128d s, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_maskz_roundscale_sd ( __m128d k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_maskz_roundscale_round_sd ( __m128d 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 Table 2-47, “Type E3 Class Exception Conditions”.
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 /r ib</td>
<td>A</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 /r ib</td>
<td>A</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 /r ib</td>
<td>A</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>A</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

$$\text{ROUND}(x) = 2^{-M}\times\text{Round} \_\text{to} \_\text{INT}(x\times2^M, \text{round} \_\text{ctrl}),$$

$$\text{round} \_\text{ctrl} = \text{imm}[3:0];$$

$$M=\text{imm}[7:4];$$

The operation of $x\times2^M$ 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 5-16.

**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])
}
```

VRNDScalePS (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF *src is a memory operand*
    THEN TMP_SRC := BROADCAST32(SRC, VL, k1)
    ELSE TMP_SRC := SRC
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := RoundToIntegerSP(TMP_SRC[i+31:i], imm8[7:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPS __m512 __mm512_roundscale_ps(__m512 a, int imm);
VRNDSCALEPS __m512 __mm512_roundscale_round_ps(__m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_ps(__mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 __mm512_maskz_roundscale_round_ps(__mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 __mm256_roundscale_ps(__m256 a, int imm);
VRNDSCALEPS __m256 __mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 __mm256_maskz_roundscale_ps(__mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 __mm128_roundscale_ps(__m128 a, int imm);
VRNDSCALEPS __m128 __mm128_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 __mm128_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 Table 2-46, "Type E2 Class Exception Conditions".
VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits

**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.LLIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 [k1] [z], xmm2, xmm3/m32{saе}, imm8</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/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 MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

\[
ROUND(x) = 2^{-M} \times \text{Round\_to\_INT}(x \times 2^{M}, \text{round\_ctrl}),
\]

\[
\text{round\_ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^{M}\) is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

\[
ROUND(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),
\]

\[
\text{round\_ctrl} = \text{imm}[3:0];
\]

**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 5-16.**
Operation

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[7:6] = 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*
    ELSE ; zeroing-masking
        THEN DEST[31:0] := 0
    FI;
FI;
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESS __m128 _mm_roundscale_ss ( __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss __mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See Table 2-47, "Type E3 Class Exception Conditions".

VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits
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>A</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>A</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>A</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>Tuple Type</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>Full</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 $\pm\infty$ 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.

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[MAXVL-1:VL] := 0

**Table 5-17. 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 = 2⁻²ⁿ</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**

VRSQRT14PD __m512d _mm512_rsqrt14_pd(__m512d a);
VRSQRT14PD __m512d _mm512_mask_rsqrt14_pd(__m512d s, __mmask8 k, __m512d a);
VRSQRT14PD __m512d _mm512_maskz_rsqrt14_pd( __mmask8 k, __m512d a);
VRSQRT14PD __m256d _mm256_rsqrt14_pd(__m256d a);
VRSQRT14PD __m256d _mm256_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d _mm256_maskz_rsqrt14_pd( __mmask8 k, __m256d a);
VRSQRT14PD __m128d _mm128_rsqrt14_pd(__m128d a);
VRSQRT14PD __m128d _mm128_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d _mm128_maskz_rsqrt14_pd( __mmask8 k, __m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions".
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.LLIG.66.0F38.W1 4F /r</td>
<td>A</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>Tuple Type</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>Tuple 1 Scalar</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 (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. When the source operand is an $+\infty$ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


Operation

**VRSQRT14SD (EVEX version)**

If $k1[0]$ or *no writemask*

IF THEN DEST[63:0] := 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] := 0

FI;

FI;

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0
Table 5-18. 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 = 2⁻²ᴷ</td>
<td>2ᴷ</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 Table 2-51, “Type E5 Class Exception Conditions”.

VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value
VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

### Instruction Set Reference, V-Z

#### Op/En/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.128.66.0F38.W0 4E /r</td>
<td>A</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>A</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>A</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>Tuple Type</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>Full</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 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.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is 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:i]));
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

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

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_rsqrt14_ps(__m128 s, __mmask8 k, __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 Table 2-21, "Type 4 Class Exception Conditions".

**Table 5-19. 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>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>
VRSQRT14SS—Compute Approximate Reciprocal of Square Root 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>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W0 4F /r</td>
<td>A</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>

**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 (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an \(\infty\) with the sign of the source value is returned. When the source operand is an \(\infty\), zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


**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* ; merging-maskning
  THEN *DEST[31:0] remains unchanged*
  ELSE ; zeroing-masking
  THEN DEST[31:0] := 0
  FI;
FI;
DEST[MAXVL-1:128] := 0
Table 5-20. 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 = 2⁻²ⁿ</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

VRSQRT14SS __m128 __m128_mm_rsqrt14_ss(__m128 a, __m128 b);
VRSQRT14SS __m128 __m128_mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRSQRT14SS __m128 __m128_mm_maskz_rsqrt14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-51, “Type E5 Class Exception Conditions”.

Vol. 2C 5-565
VSCALEFPD—Scale Packed Float64 Values With Float64 Values

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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 them by 2 to the power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

\[ z_{mm1} = z_{mm2} \times 2^{\text{floor}(z_{mm3})}. \]

Floor(z_{mm3}) means maximum integer value \( \leq z_{mm3} \).

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 5-21 and Table 5-22.

**Table 5-21. VSCALEFPD/SD/PS/SS Special Cases**

<table>
<thead>
<tr>
<th>Src1</th>
<th>Src2</th>
<th>0/Denorm/Normal</th>
<th>Set IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>±NaN</td>
<td>±Inf</td>
<td>+0</td>
<td>QNaN(Src1) IF either source is SNAN</td>
</tr>
<tr>
<td>±SNAN</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1) YES</td>
</tr>
<tr>
<td>±Inf</td>
<td>QNaN(Src2)</td>
<td>Src1</td>
<td>QNaN_Indefinite Src1 IF Src2 is SNAN or -INF</td>
</tr>
<tr>
<td>±0</td>
<td>QNaN(Src2)</td>
<td>QNaN_Indefinite Src1</td>
<td>Src1 IF Src2 is SNAN or +INF</td>
</tr>
<tr>
<td>Denorm/Normal</td>
<td>QNaN(Src2)</td>
<td>±INF (Src1 sign)</td>
<td>±0 (Src1 sign) Compute Result IF Src2 is SNAN</td>
</tr>
</tbody>
</table>

**Table 5-22. VSCALEFPD/VS/VD/VP Special Cases**

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 2C /r</td>
<td>VSCALEFPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 2C /r</td>
<td>VSCALEFPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 2C /r</td>
<td>VSCALEFPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
</tr>
</tbody>
</table>

**Instruction Set Reference, V-Z**

5-566 Vol. 2C
Table 5-22. 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>&lt; 2^{-1074}</td>
</tr>
<tr>
<td>(</td>
<td>result</td>
<td>≥ 2^{1024}</td>
</tr>
</tbody>
</table>

**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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR \(j := 0\) TO \(KL-1\)
\(i := j \times 64\)
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[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d _mm512_scalef_pd(__m512d a, __m512d b);
VSCALEFPD __m512d _mm512_mask_scalef_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m512d _mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m256d _mm256_scalef_pd(__m256d a, __m256d b);
VSCALEFPD __m256d _mm256_mask_scalef_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m256d _mm256_maskz_scalef_pd(__mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m128d _mm_scalef_pd(__m128d a, __m128d b);
VSCALEFPD __m128d _mm_mask_scalef_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VSCALEFPD __m128d _mm_maskz_scalef_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions
See Table 2-46, “Type E2 Class Exception Conditions”.
**VSALFED—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.LLIG.66.0F38.W1 2D /r</td>
<td>A</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>Tuple Type</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>Tuple1 Scalar</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 double-precision floating-point value in the first source operand by multiplying it by 2 to the power of the double-precision floating-point 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 ≤ 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 5-21 and Table 5-22.

**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] * PW[2, Floor(TMP_SRC2[63:0])]
}
```
VSACLEFSD (EVEX encoded version)

IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
    FI;
IF k1[0] OR *no writemask*
    THEN DEST[63:0] := SCALE(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[63:0] := 0
        FI
    FI
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VSACLEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSACLEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSACLEFSD __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 Table 2-47, “Type E3 Class Exception Conditions”.

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions”.
**VSACLEFPS—Scale Packed Float32 Values With 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 2C /r</td>
<td>A V/V</td>
<td>AVX512VL</td>
<td>Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 2C /r</td>
<td>A V/V</td>
<td>AVX512VL</td>
<td>Scale the packed single-precision values in ymm2 using floating point values from ymm3/m256/m32bcst. Under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 2C /r</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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 single-precision floating-point values in the first source operand by multiplying them by 2 to the power of the float32 values in second source operand.

The equation of this operation is given by:

$$ z_{mm1} := z_{mm2} \times 2^{\text{floor}(z_{mm3})}. $$

Floor(z_{mm3}) means maximum integer value ≤ z_{mm3}.

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 5-21 and Table 5-23.

**Table 5-23. Additional VSACLEFPS/SS Special Cases**

<table>
<thead>
<tr>
<th>Special Case</th>
<th>Returned value</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>result</td>
<td>&lt; 2^{-149}$</td>
</tr>
<tr>
<td>$</td>
<td>result</td>
<td>\geq 2^{-128}$</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_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
i := j * 32
IF k1[j] OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[31:0]);
ELSE DEST[i+31:i] := SCALE(SRC1[i+31:i], SRC2[i+31:i]);
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+31:i] := 0
FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0;

Intel C/C++ Compiler Intrinsic Equivalent
VSCALEFPS __m512 __mm512_scalef_round_ps(__m512 a, __m512 b, int rounding);
VSCALEFPS __m512 __mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 __mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int rounding);
VSCALEFPS __m512 __mm512_scalef_ps(__m512 a, __m512 b);
VSCALEFPS __m512 __mm512_mask_scalef_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m512 __mm512_maskz_scalef_ps(__mmask16 k, __m512 a, __m512 b);
VSCALEFPS __m256 __mm256_scalef_ps(__m256 a, __m256 b);
VSCALEFPS __m256 __mm256_mask_scalef_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m256 __mm256_maskz_scalef_ps(__mmask8 k, __m256 a, __m256 b);
VSCALEFPS __m128 __mm128_scalef_ps(__m128 a, __m128 b);
VSCALEFPS __m128 __mm128_mask_scalef_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VSCALEFPS __m128 __mm128_maskz_scalef_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions
Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.
Other Exceptions
See Table 2-46, "Type E2 Class Exception Conditions".
VSCALEFSS—Scale Scalar Float32 Value With 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.LLIG.66.0F38.W0 2D /r</td>
<td>A</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>Tuple Type</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>Tuple 1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMr/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 to the 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 ≤ 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 5-21 and Table 5-23.
**Operation**

**SCALE(SRC1, SRC2)**

```plaintext
{ 
    ; 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*

```plaintext
THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
```

```plaintext
IF k[1][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;
```

```plaintext
DEST[MAXVL-1:128] := 0
```

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

```plaintext
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 Table 2-47, “Type E3 Class Exception Conditions”. 
VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

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.

<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>VSCATTERDPS vm32x {k1}, xmm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERDPS vm32y {k1}, ymm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERDPS vm32z {k1}, zmm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERDPD vm32x {k1}, xmm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERDPD vm32y {k1}, ymm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERDPD vm32z {k1}, zmm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERQPS vm64x {k1}, xmm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERQPS vm64y {k1}, ymm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERQPS vm64z {k1}, zmm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERQPD vm64x {k1}, xmm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERQPD vm64y {k1}, ymm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>VSCATTERQPD vm64z {k1}, zmm1</td>
<td>A/V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Tuple1 Scalar</td>
<td>BaseReg(R): VSIB:base, VectorReg(R): VSIB: index</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
• 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 scattered 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 scattered.
• 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 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 * 64
    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

VSCATTERQPD void _mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERQPS void _mm512_i32scatter_ps(void * base, __m512i vdx, __m256 a, int scale);
VSCATTERQPS void _mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m256 a, int scale);
VSCATTERQPD void _mm512_i64scatter_pd(void * base, __m256i 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, __m256i vdx, __m256 a, int scale);
VSCATTERQPS void _mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void _mm256_i32scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERQPD void _mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m256d a, int scale);
VSCATTERQPS void _mm256_i32scatter_ps(void * base, __m128i vdx, __m256 a, int scale);
VSCATTERQPS void _mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m256 a, int scale);
VSCATTERQPD void _mm256_i64scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERQPD void _mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m256d a, int scale);
VSCATTERQPS void _mm256_i64scatter_ps(void * base, __m128i vdx, __m256 a, int scale);
VSCATTERQPS void _mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m256 a, int scale);
**SIMD Floating-Point Exceptions**
Invalid, Overflow, Underflow, Precision, Denormal

**Other Exceptions**
See Table 2-61, “Type E12 Class Exception Conditions”.
## VSHUFF32x4/VSHUFF64x2/VSHUF32x4/VSHUF64x2—Shuffle Packed Values at 128-bit Granularity

<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.256.66.0F3A.w0 23 / r ib VSHUFF32x4 ymm1<a href="z">k1</a>, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.w0 23 / r ib VSHUFF32x4 zmm1<a href="z">k1</a>, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.w1 23 / r ib VSHUFF64x2 ymm1<a href="z">k1</a>, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.w1 23 / r ib VSHUFF64x2 zmm1<a href="z">k1</a>, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.w0 43 / r ib VSHUF32x4 ymm1<a href="z">k1</a>, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.w0 43 / r ib VSHUF32x4 zmm1<a href="z">k1</a>, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.w1 43 / r ib VSHUF64x2 ymm1<a href="z">k1</a>, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.w1 43 / r ib VSHUF64x2 zmm1<a href="z">k1</a>, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple Type</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>Full</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[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                THEN DEST[i+31:i] := 0
            FI;
        FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VSHUFF64x2 (EVEX 512-bit version)
(KL, VL) = (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] := SRC2[63:0]
    ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[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
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VSHUF32x4 (EVEX 512-bit version)
(KL, VL) = (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] := SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
  i := j * 32
IF k1[j] OR *no writemask*
THEN DEST[i+31:i] := TMP_DEST[i+31:i]
ELSE
  IF *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
  ELSE *zeroing-masking* ; zeroing-masking
    THEN DEST[i+31:i] := 0
  FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VSHUF64x2 (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[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        THEN DEST[i+63:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VSHUFF32x4/VSHUFF64x2/VSHUF32x4/VSHUF64x2—Shuffle Packed Values at 128-bit Granularity

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFI32x4 __m512i _mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m512 __mm512_shuffle_f32x4(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 __mm512_mask_shuffle_f32x4(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 __mm512_maskz_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 __mm512_shuffle_f64x2(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 __mm512_mask_shuffle_f64x2(__m512 s, __mmask8 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 __mm512_maskz_shuffle_f64x2(__mmask8 k, __m512 a, __m512 b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-50, “Type E4NF Class Exception Conditions”; additionally:

#UD If EVEX.L’L = 0 for VSHUFF32x4/VSHUFF64x2.
### Instruction Set Reference, V-Z

#### VTESTPD/VTESTPS—Packed Bit Test

<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 VTESTPS xmm1, xmm2/m128</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0E /r VTESTPS ymm1, ymm2/m256</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 0F /r VTESTPD xmm1, xmm2/m128</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0F /r VTESTPD ymm1, ymm2/m256</td>
<td>RM 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>
<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>

### Description

VTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M reg field.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

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

**VTESTPS (128-bit version)**
\[\text{TEMP}[127:0] := \text{SRC}[127:0] \text{ AND } \text{DEST}[127:0]\]
IF (\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = 0)
  THEN ZF := 1;
  ELSE ZF := 0;
\[\text{TEMP}[127:0] := \text{SRC}[127:0] \text{ AND NOT } \text{DEST}[127:0]\]
IF (\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = 0)
  THEN CF := 1;
  ELSE CF := 0;
\text{DEST (unmodified)}
\text{AF := OF := PF := SF := 0;}

**VTESTPS (VEX.256 encoded version)**
\[\text{TEMP}[255:0] := \text{SRC}[255:0] \text{ AND } \text{DEST}[255:0]\]
IF (\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = 0)
  THEN ZF := 1;
  ELSE ZF := 0;
\[\text{TEMP}[255:0] := \text{SRC}[255:0] \text{ AND NOT } \text{DEST}[255:0]\]
IF (\text{TEMP}[31] = \text{TEMP}[63] = \text{TEMP}[95] = \text{TEMP}[127] = 0)
  THEN CF := 1;
  ELSE CF := 0;
\text{DEST (unmodified)}
\text{AF := OF := PF := SF := 0;}

**VTESTPD (128-bit version)**
\[\text{TEMP}[127:0] := \text{SRC}[127:0] \text{ AND } \text{DEST}[127:0]\]
IF (\text{TEMP}[63] = \text{TEMP}[127] = 0)
  THEN ZF := 1;
  ELSE ZF := 0;
\[\text{TEMP}[127:0] := \text{SRC}[127:0] \text{ AND NOT } \text{DEST}[127:0]\]
IF (\text{TEMP}[63] = \text{TEMP}[127] = 0)
  THEN CF := 1;
  ELSE CF := 0;
\text{DEST (unmodified)}
\text{AF := OF := PF := SF := 0;}

**VTESTPD (VEX.256 encoded version)**
\[\text{TEMP}[255:0] := \text{SRC}[255:0] \text{ AND } \text{DEST}[255:0]\]
IF (\text{TEMP}[63] = \text{TEMP}[127] = \text{TEMP}[191] = \text{TEMP}[255] = 0)
  THEN ZF := 1;
  ELSE ZF := 0;
\[\text{TEMP}[255:0] := \text{SRC}[255:0] \text{ AND NOT } \text{DEST}[255:0]\]
IF (\text{TEMP}[63] = \text{TEMP}[127] = \text{TEMP}[191] = \text{TEMP}[255] = 0)
  THEN CF := 1;
  ELSE CF := 0;
\text{DEST (unmodified)}
\text{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 Table 2-21, "Type 4 Class Exception Conditions"; additionally:

- **#UD**
  - If VEX.vvvv ≠ 1111B.
  - If VEX.W = 1 for VTESTPS or VTESTPD.
VZEROALL—Zero XMM, YMM and ZMM Registers

<table>
<thead>
<tr>
<th>Op/En 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>VZEROALL</td>
<td>ZO</td>
<td>V/V</td>
<td>AVX</td>
<td>Zero some of the XMM, YMM and ZMM registers.</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>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

In 64-bit mode, the instruction zeroes XMM0-XMM15, YMM0-YMM15, and ZMM0-ZMM15. Outside 64-bit mode, it zeroes only XMM0-XMM7, YMM0-YMM7, and ZMM0-ZMM7. VZEROALL does not modify ZMM16-ZMM31.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

**Operation**

`simd_reg_file[]` is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0 and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512.

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

IF (64-bit mode)
  limit := 15
ELSE
  limit := 7
FOR i in 0 .. limit:
  simd_reg_file[i][MAXVL-1:0] := 0

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

VZEROALL: 

```c
_mm256_zeroall()
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-25, “Type 8 Class Exception Conditions”. 

VZEROUPPER—Zero Upper Bits of YMM and ZMM Registers

<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.128.0F.WIG 77 VZEROUPPER</td>
<td>ZO</td>
<td>V/V</td>
<td>AVX</td>
<td>Zero bits in positions 128 and higher of some YMM and ZMM registers.</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>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

In 64-bit mode, the instruction zeroes the bits in positions 128 and higher in YMM0-YMM15 and ZMM0-ZMM15. Outside 64-bit mode, it zeroes those bits only in YMM0-YMM7 and ZMM0-ZMM7. VZEROUPPER does not modify the lower 128 bits of these registers and it does not modify ZMM16-ZMM31.

This instruction is recommended when transitioning between AVX and legacy SSE code; it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

**Operation**

simd_reg_file[][] is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0 and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits.

```
VZEROUPPER
IF (64-bit mode)
    limit := 15
ELSE
    limit := 7
FOR i in 0 .. limit:
    simd_reg_file[i][MAXVL-1:128] := 0
```

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

VZEROUPPER: _mm256_zeroupper()

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-25, "Type 8 Class Exception Conditions".
WAIT/FWAIT—Wait

### 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.

---

<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>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Check pending unmasked floating-point exceptions.</td>
</tr>
<tr>
<td>9B</td>
<td>FWAIT</td>
<td>ZO</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>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
WBINVD—Write Back and Invalidate Cache

### Opcode

<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 09</td>
<td>WBINVD</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Write back and flush internal caches; initiate writing-back and flushing of external caches.</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>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### 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 *)

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

`WBINVD void _wbinvd(void);`

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If the current privilege level is not 0.
- #UD If the LOCK prefix is used.
Real-Address Mode Exceptions
#UD If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) 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.
**WBNOINVD—Write Back and Do Not Invalidate Cache**

<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>F3 0F 09</td>
<td>ZO</td>
<td>V/V</td>
<td>WBNOINVD</td>
<td>Write back and do not flush internal caches; initiate writing-back without flushing of external caches.</td>
</tr>
<tr>
<td>WBNOINVD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Tuple</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

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

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

The WBNOINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see “Serializing Instructions” in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A).

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

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

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

WBNOINVD void _wbnoinvd(void);

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0) If the current privilege level is not 0.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#UD If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) WBNOINVD cannot be executed at the virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.
64-Bit Mode Exceptions
Same exceptions as in protected mode.
**WRFSBASE/WRGSBASE—Write FS/GS Segment Base**

<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>F3 0F AE /2 WRFSBASE r32</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the FS base address with the 32-bit value in the source register.</td>
</tr>
<tr>
<td>F3 REX.W 0F AE /2 WRFSBASE r64</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the FS base address with the 64-bit value in the source register.</td>
</tr>
<tr>
<td>F3 0F AE /3 WRGSBASE r32</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the GS base address with the 32-bit value in the source register.</td>
</tr>
<tr>
<td>F3 REX.W 0F AE /3 WRGSBASE r64</td>
<td>M</td>
<td>V/I</td>
<td>FSGSBASE</td>
<td>Load the GS base address with the 64-bit value in the source 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>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.
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>Comp/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 30</td>
<td>WRMSR</td>
<td>ZO</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>ZO</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 2, “Model-Specific Registers (MSRs)” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

**IA-32 Architecture Compatibility**

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

**Operation**

\[
\text{MSR(ECX)} := \text{EDX:EAX};
\]

**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/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>NP 0F 01 EF WRPKR</td>
<td>ZO</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>ZO</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 WRPKR 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.

WRPKRU will never execute speculatively. Memory accesses affected by PKRU register will not execute (even speculatively) until all prior executions of WRPKR have completed execution and updated the PKRU register.

Operation

IF (ECX = 0 AND EDX = 0)
    THEN PKRU := EAX;
ELSE #GP(0);
FI;

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

```c
WRPKRU: void _wrpkru(uint32_t);
```

Protected Mode Exceptions

- #GP(0)       If ECX ≠ 0.
- #UD           If the LOCK prefix is used.
- #UD           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.
WRSSD/WRSSQ—Write to Shadow Stack

<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>OF 38 F6 l(11):rrr:bbb WRSSD m32, r32</td>
<td>MR</td>
<td>V/V</td>
<td>CET_SS</td>
<td>Write 4 bytes to shadow stack.</td>
</tr>
<tr>
<td>REX.W OF 38 F6 l(11):rrr:bbb WRSSQ m64, r64</td>
<td>MR</td>
<td>V/N.E.</td>
<td>CET_SS</td>
<td>Write 8 bytes to shadow stack.</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>MR</td>
<td>ModRM:r/m(w)</td>
<td>ModRM:reg(r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

Writes bytes in register source to the shadow stack.

Operation

IF CPL = 3
  IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
    THEN #UD; Fl;
  IF (IA32_U_CET.WR_SHSTK_EN) = 0
    THEN #UD; Fl;
ELSE
  IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
    THEN #UD; Fl;
  IF (IA32_S_CET.WR_SHSTK_EN) = 0
    THEN #UD; Fl;
FI;
DEST_LA = Linear_Address(mem operand)
IF (operand size is 64 bit)
  THEN
    (* Destination not 8B aligned *)
    IF DEST_LA[2:0]
      THEN GP(0); Fl;
      Shadow_stack_store 8 bytes of SRC to DEST_LA;
    ELSE
      (* Destination not 4B aligned *)
      IF DEST_LA[1:0]
        THEN GP(0); Fl;
        Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA;
  Fl;

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRSSD   void _wrssd(__int32, void *);
WRSSQ   void _wrssq(__int64, void *);
Protected Mode Exceptions
#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
   If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
   If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
   If destination is located in a non-writeable segment.
   If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment
   selector.
   If linear address of destination is not 4 byte aligned.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor
shadow stack when CPL < 3.
   Other terminal and non-terminal faults.

Real-Address Mode Exceptions
#UD The WRSS instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions
#UD The WRSS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions
#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
   If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
   If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.

#PF(fault-code) If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor
shadow stack when CPL < 3.
   Other terminal and non-terminal faults.

64-Bit Mode Exceptions
#UD If the LOCK prefix is used.
   If CR4.CET = 0.
   If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
   If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
   If CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
   If CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.

#GP(0) If a memory address is in a non-canonical form.
   If linear address of destination is not 4 byte aligned.

#PF(fault-code) If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor
shadow stack when CPL < 3.
   Other terminal and non-terminal faults.
WRUSSD/WRUSSQ—Write to User Shadow Stack

<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 38 F5 l(11):rrr:bbb WRUSSD m32, r32</td>
<td>MR</td>
<td>V/V</td>
<td>CET_SS</td>
<td>Write 4 bytes to shadow stack.</td>
</tr>
<tr>
<td>66 REX W 0F 38 F5 l(11):rrr:bbb WRUSSQ m64, r64</td>
<td>MR</td>
<td>V/N.E.</td>
<td>CET_SS</td>
<td>Write 8 bytes to shadow stack.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Opernd Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>MR</td>
</tr>
</tbody>
</table>

**Description**
Writs bytes in register source to a user shadow stack page. The WRUSS instruction can be executed only if CPL = 0, however the processor treats its shadow-stack accesses as user accesses.

**Operation**
IF CR4.CET = 0
  THEN #UD; Fi;
IF CPL > 0
  THEN #GP(0); Fi;
DEST_LA = Linear_Address(mem operand)
IF (operand size is 64 bit)
  THEN
    (* Destination not 8B aligned *)
    IF DEST_LA[2:0]
      THEN GP(0); Fi;
      Shadow_stack_store 8 bytes of SRC to DEST_LA as user-mode access;
    ELSE
      (* Destination not 4B aligned *)
      IF DEST_LA[1:0]
        THEN GP(0); Fi;
        Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA as user-mode access;
  ELSE
  FI;

**Flags Affected**
None.

**C/C++ Compiler Intrinsic Equivalent**
WRUSSD void _wrussd(__int32, void *);
WRUSSQ void _wrussq(__int64, void *);
Protected Mode Exceptions

#UD If the LOCK prefix is used.
If CR4.CET = 0.

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If destination is located in a non-writeable segment.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
If linear address of destination is not 4 byte aligned.
If CPL is not 0.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If destination is not a user shadow stack.
Other terminal and non-terminal faults.

Real-Address Mode Exceptions

#UD The WRUSS instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.
If CR4.CET = 0.

#GP(0) If a memory address is in a non-canonical form.
If linear address of destination is not 4 byte aligned.
If CPL is not 0.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If destination is not a user shadow stack.
Other terminal and non-terminal faults.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If CR4.CET = 0.

#GP(0) If a memory address is in a non-canonical form.
If linear address of destination is not 4 byte aligned.
If CPL is not 0.

#PF(fault-code) If destination is not a user shadow stack.
Other terminal and non-terminal faults.
**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 *)
      Fl; (* 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
      Fl;
   ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)
      GOTO HLE_ABORT_PROCESSING
   FI;
ELSE
   Treat instruction as non-XACQUIRE F2H prefixed legacy instruction
FI;

F2H
F3H
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;

(* 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>Operand 1</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**

\[
\begin{align*}
\text{IF } \text{RTM_ACTIVE} &= 0 \\
\text{THEN} &
\begin{align*}
\text{Treat as NOP;} \\
\text{ELSE} &
\begin{align*}
\text{GOTO RTM_ABORT_PROCESSING;} \\
\text{FI;
\end{align*}
\end{align*}
\end{align*}
\]

(* 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
  \[
  \begin{align*}
  \text{THEN} &
  \begin{align*}
  \text{RIP:= fallbackRIP;} \\
  \text{ELSE} &
  \begin{align*}
  \text{EIP := fallbackEIP;} \\
  \text{FI;
  \end{align*}
  \end{align*}
  \end{align*}
  \]
- END

### Flags Affected

None

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

XABORT: `void _xabort(unsigned int);`

### SIMD Floating-Point Exceptions

None
Other Exceptions

#UD

CPUID. (EAX = 7, ECX = 0): EBX.RTM[bit 11] = 0.
If LOCK prefix is used.
XADD—Exchange and Add

Op/W  Instruction  Op/ En  64-Bit Mod/ Description

OF CO /r  XADD r/m8, r8  MR  Valid  Valid  Exchange r8 and r/m8; load sum into r/m8.
REX + OF CO /r  XADD r/m8, r8*  MR  Valid  N.E.  Exchange r8 and r/m8; load sum into r/m8.
OF CI /r  XADD r/m16, r16  MR  Valid  Valid  Exchange r16 and r/m16; load sum into r/m16.
OF CI /r  XADD r/m32, r32  MR  Valid  Valid  Exchange r32 and r/m32; load sum into r/m32.
REX.W + OF CI /r  XADD r/m64, r64  MR  Valid  N.E.  Exchange r64 and r/m64; load sum into r/m64.

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

Instruction Operand Encoding

<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>

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

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

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

IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

Operation

TEMP := SRC + DEST;
SRC := DEST;
DEST := TEMP;

Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

Protected Mode Exceptions

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

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

<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>Operand2</th>
<th>Operand3</th>
<th>Operand4</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. (Use of the 16-bit operand size does not cause this address to be truncated to 16 bits, unlike a near jump to a relative offset.)

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**

```plaintext
XBEGIN
IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
  THEN
    RTM_NEST_COUNT++
    IF RTM_NEST_COUNT = 1 THEN
      IF 64-bit Mode
        THEN
          IF OperandSize = 16
            THEN fallbackRIP := RIP + SignExtend64(rel16);
            ELSE fallbackRIP := RIP + SignExtend64(rel32);
            FI;
          IF fallbackRIP is not canonical
            THEN #GP(0);
            FI;
        ELSE
          IF OperandSize = 16
            THEN fallbackEIP := EIP + SignExtend32(rel16);
            ELSE fallbackEIP := EIP + rel32;
            FI;
          IF fallbackEIP outside code segment limit
            THEN #GP(0);
            FI;
        ENDIF;
      RTM_ACTIVE := 1
      Enter RTM Execution (* record register state, start tracking memory state*)
    ENDIF;
  ENDIF;
ENDIF;
```

XBEGIN — Transactional Begin
INSTRUCTION SET REFERENCE, V-Z

FI; (* RTM_NEST_COUNT = 1 *)
ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
   GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
   Restore architectural register state
   Discard memory updates performed in transaction
   Update EAX with status
   RTM_NEST_COUNT := 0
   RTM_ACTIVE := 0
   IF 64-bit mode
      THEN
         RIP := fallbackRIP
      ELSE
         EIP := fallbackEIP
   FI;
END

Flags Affected
None

Intel C/C++ Compiler Intrinsic Equivalent
XBEGIN: unsigned int _xbegin( void );

SIMD Floating-Point Exceptions
None

Protected Mode Exceptions
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
   If LOCK prefix is used.
#GP(0) If the fallback address is outside the CS segment.

Real-Address Mode Exceptions
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
   If LOCK prefix is used.

Virtual-8086 Mode Exceptions
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
   If LOCK prefix is used.

Compatibility Mode Exceptions
Same exceptions as in protected mode.
64-bit Mode Exceptions
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
   If LOCK prefix is used.
#GP(0) If the fallback address is non-canonical.
XCHG—Exchange Register/Memory with Register

<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>90+rw</td>
<td>XCHG AX, r16</td>
<td>0</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r16 with AX.</td>
</tr>
<tr>
<td>90+rw</td>
<td>XCHG AX, AX</td>
<td>0</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>0</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>0</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange r64 with RAX.</td>
</tr>
<tr>
<td>90+rd</td>
<td>XCHG r32, EAX</td>
<td>0</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>0</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange RAX with r64.</td>
</tr>
<tr>
<td>86 /r</td>
<td>XCHG r/m8, r8</td>
<td>MR</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange 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>Valid</td>
<td>N.E.</td>
<td>Exchange byte register with byte from r/m8.</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>Valid</td>
<td>N.E.</td>
<td>Exchange byte from r/m8 with r8 (byte register).</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>Valid</td>
<td>N.E.</td>
<td>Exchange r64 with quadword from r/m64.</td>
</tr>
<tr>
<td>87 /r</td>
<td>XCHG r32, r/m32</td>
<td>RM</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange quadword from r/m32 with r32.</td>
</tr>
<tr>
<td>REX.W + 87 /r</td>
<td>XCHG r64, r/m64</td>
<td>RM</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange quadword from r/m64 with r64.</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:reg (r)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:reg (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.
 If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
 If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used but the destination is not a memory operand.

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

Virtual-8086 Mode Exceptions
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made.
#UD If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

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

<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>NP 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>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</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;  
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 prefix is used.

- **#GP(0)**
  If RTM_ACTIVE = 0.
### XGETBV—Get Value of Extended Control Register

<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>NP 0F 01 D0</td>
<td>XGETBV</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Reads an XCR specified by ECX into EDX:EAX.</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>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

#### Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, “Managing State Using the XSAVE Feature Set,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Use of any other value for ECX results in a general-protection (#GP) exception.

#### Operation

EDX:EAX := XCR[ECX];

#### Flags Affected

None.

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

XGETBV: unsigned __int64 _xgetbv(unsigned int);

#### Protected Mode Exceptions

- **#GP(0)**: If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- **#UD**: If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

#### Real-Address Mode Exceptions

- **#GP(0)**: If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- **#UD**: If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

#### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions
Same exceptions as in protected mode.
**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-operators 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
    \[ AL := (DS:BX + \text{ZeroExtend}(AL)) \];
    ELSE IF (AddressSize = 32)
        \[ AL := (DS:EBX + \text{ZeroExtend}(AL)) \]; Ff;
    ELSE (AddressSize = 64)
        \[ AL := (RBX + \text{ZeroExtend}(AL)) \];
    FI;
FI;

**Flags Affected**

None.

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)** If the DS, ES, FS, or GS register contains a NULL segment selector.
- **#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

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.

Instruction Operand 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 id</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 id</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.
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**

\[ \text{DEST} := \text{DEST} \text{ XOR } \text{SRC} ; \]

**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.
- **#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.
**XORPD—Bitwise Logical XOR 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>66 0F 57/r XORPD xmm1, xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>SSE2</td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in xmm1 and xmm2/mem.</td>
</tr>
<tr>
<td>VEX.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/mem.</td>
</tr>
<tr>
<td>VEX.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256</td>
<td>B</td>
<td>V/V</td>
<td>AVX</td>
<td>Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem.</td>
</tr>
<tr>
<td>EVEX.128.66.0F.W1 57 /r VXORPD xmm1 (k1){z}, xmm2, xmm3/m128/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</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.256.66.0F.W1 57 /r VXORPD ymm1 (k1){z}, ymm2, ymm3/m256/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</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.512.66.0F.W1 57 /r VXORPD zmm1 (k1){z}, zmm2, zmm3/m512/m64bcst</td>
<td>C</td>
<td>V/V</td>
<td>AVX512DQ</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>Tuple Type</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>NA</td>
<td>ModRM[reg (r, w)]</td>
<td>ModRM[r/m (r)]</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM[reg (w)]</td>
<td>VEX.vvvv (r)</td>
<td>ModRM[r/m (r)]</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</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 (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 and EVEX.128 encoded versions:** The first source operand is an XMM register. The second source operand is an XMM register or a 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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

**VXORPD (EVEX encoded versions)**

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

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

DEST[MAXVL-1:VL] := 0

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

DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[MAXVL-1:256] := 0

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

DEST[63:0] := SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[MAXVL-1:128] := 0

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

DEST[63:0] := DEST[63:0] BITWISE XOR SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

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

VXORPD __m512d _mm512_xor_pd (__m512d a, __m512d b);
VXORPD __m512d _mm512_mask_xor_pd (__m512d a, __mmask8 m, __m512d b);
VXORPD __m512d _mm512_maskz_xor_pd (__mmask8 m, __m512d a);
VXORPD __m256d _mm256_xor_pd (__m256d a, __m256d b);
VXORPD __m256d _mm256_mask_xor_pd (__m256d a, __mmask8 m, __m256d b);
VXORPD __m256d _mm256_maskz_xor_pd (__mmask8 m, __m256d a);
XORPD __m128d _mm_xor_pd (__m128d a, __m128d b);
VXORPD __m128d _mm_mask_xor_pd (__m128d a, __mmask8 m, __m128d b);
VXORPD __m128d _mm_maskz_xor_pd (__mmask8 m, __m128d a);

**SIMD Floating-Point Exceptions**

None
Other Exceptions

Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions”. 
### 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>NP 0F 57 /r</td>
<td>A</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.128.0F:W1G 57 /r</td>
<td>B</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.256.0F:W1G 57 /r</td>
<td>B</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.128.0F:W0 57 /r</td>
<td>C</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.256.0F:W0 57 /r</td>
<td>C</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.512.0F:W0 57 /r</td>
<td>C</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>Tuple Type</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>NA</td>
<td>ModRM:reg (r, w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>NA</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>C</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

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

**EVEX.512 encoded version:** The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

**VEX.256 and EVEX.256 encoded versions:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 and EVEX.128 encoded versions:** The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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

**VXORPS (EVEX encoded versions)**

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

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

DEST[MAXVL-1:VL] := 0
```

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

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[MAXVL-1:256] := 0
```

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

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[MAXVL-1:128] := 0
```

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

```
DEST[31:0] := SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[95:64] := SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)
```

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

```
VXORPS __m512 _mm512_xor_ps (__m512 a, __m512 b);
VXORPS __m512 __mm512_mask_xor_ps (__m512 a, __mmask16 m, __m512 b);
VXORPS __m512 __mm512_maskz_xor_ps (__mmask16 m, __m512 a);
VXORPS __m256 __mm256_xor_ps (__m256 a, __m256 b);
VXORPS __m256 __mm256_mask_xor_ps (__m256 a, __mmask8 m, __m256 b);
VXORPS __m256 __mm256_maskz_xor_ps (__mmask8 m, __m256 a);
XORPS __m128 __mm_xor_ps (__m128 a, __m128 b);
VXORPS __m128 __mm_mask_xor_ps (__m128 a, __mmask8 m, __m128 b);
```
VXORPS __m128 __mm_maskz_xor_ps (__mmask8 m, __m128 a);

**SIMD Floating-Point Exceptions**
None

**Other Exceptions**
Non-EVEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-49, “Type E4 Class Exception Conditions”.
XRSTOR—Restore Processor Extended States

<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>NP 0F AE /5 XRSTOR mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVE</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
<tr>
<td>NP REX.W + 0F AE /5 XRSTOR64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSAVE</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>ModRMr/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. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” 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 1.\(^1\)

• If RFBM[1] = 1 and bit 1 is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component 1.

• If RFBM[1] = 1 and XSTATE_BV[1] = 1, XRSTOR loads state component 1 from the XSAVE area.

• The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.

• XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.

• Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component i for which RFBM[i] = 1 and XSTATE_BV[i] = 1; it tracks as modified any state component i for which RFBM[i] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

---

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.
Operation

RFBM := XCR0 AND EDX:EAX; /* bitwise logical AND */
COMPMASK := XCOMP_BV field from XSAVE header;
RSTORMASK := XSTATE_BV field from XSAVE header;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        TO_BE_RESTORED := RFBM AND RSTORMASK;
        TO_BE_INITIALIZED := RFBM AND NOT RSTORMASK;

        IF TO_BE_RESTORED[0] = 1
            THEN
                XINUSE[0] := 1;
                load x87 state from legacy region of XSAVE area;
            ELSIF TO_BE_INITIALIZED[0] = 1
                THEN
                    XINUSE[0] := 0;
                    initialize x87 state;
            FI;

            THEN load MXCSR from legacy region of XSAVE area;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                XINUSE[1] := 1;
                load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR
            ELSIF TO_BE_INITIALIZED[1] = 1
                THEN
                    XINUSE[1] := 0;
                    set all XMM registers to 0; // this step does not initialize MXCSR
            FI;

        FOR i := 2 TO 62
            IF TO_BE_RESTORED[i] = 1
                THEN
                    XINUSE[i] := 1;
                    load XSAVE state component i at offset n from base of XSAVE area;
                    // n enumerated by CPUID(EAX=0DH,ECX=i):EBX)
            ELSIF TO_BE_INITIALIZED[i] = 1
                THEN
                    XINUSE[i] := 0;
                    initialize XSAVE state component i;
            FI;
        ENDFOR;

    ELSE
        /* Compacted form of XRSTOR */
        IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
            THEN /* compacted form not supported */
                #GP(0);
        FI;

ENDIF;
FORMAT = COMPMASK AND 7FFFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED := RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT := RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
  THEN
    XINUSE[0] := 1;
    load x87 state from legacy region of XSAVE area;
ELSIF TO_BE_INITIALIZED[0] = 1
  THEN
    XINUSE[0] := 0;
    initialize x87 state;
FI;

IF TO_BE_RESTORED[1] = 1
  THEN
    XINUSE[1] := 1;
    load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
ELSIF TO_BE_INITIALIZED[1] = 1
  THEN
    set all XMM registers to 0;
    XINUSE[1] := 0;
    MXCSR := 1F80H;
FI;

NEXT_FEATURE_OFFSET = 576; // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
  IF FORMAT[i] = 1
    THEN
      IF TO_BE_RESTORED[i] = 1
        THEN
          load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
        FI;
      NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
      FI;
    IF TO_BE_INITIALIZED[i] = 1
      THEN
        XINUSE[i] := 0;
        initialize XSAVE state component i;
      FI;
  ENDFOR;
FI;

XMODIFIED := NOT RFBM;

IF in VMX non-root operation
  THEN VMXNR := 1;
  ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
XRSTOR_INFO := (CPL,VMXNR,LAXA,COMPMASK);

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XRSTOR: void _xrstor( void *, unsigned __int64);
XRSTOR: void _xrstor64( void *, unsigned __int64);

Protected Mode Exceptions

#GP(0) 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 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 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 CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.
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 the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**
Same exceptions as in protected mode

**Compatibility Mode Exceptions**
Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0) If 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 the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-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

<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>NP 0F C7 /3 XRSTORS</td>
<td>M</td>
<td>V/V</td>
<td>XSS</td>
<td>Restore state components specified by EDX:EAX from mem.</td>
</tr>
<tr>
<td>NP REX.W + 0F C7 /3</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSS</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 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. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” 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” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[i] = 0, XRSTORS does not update state component i.
- If RFBM[i] = 1 and bit i is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component i.
- If RFBM[i] = 1 and XSTATE_BV[i] = 1, XRSTORS loads state component i from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component i for which RFBM[i] = 1 and XSTATE_BV[i] = 1; it tracks as modified any state component i for which RFBM[i] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.
Operation

RFBM := (XCR0 or IA32_XSS) AND EDX:EAX; /* bitwise logical OR and AND */
COMPmask := XCOMP_BV field from XSAVE header;
RSTORMask := XSTATE_BV field from XSAVE header;

FORMAT = COMPMask AND 7FFFFFFF_FFFFFFFFH;
RESTORE FEATURES := FORMAT AND RFBM;
TO_BE_RESTORED := RESTORE FEATURES AND RSTORMask;
FORCE_INIT := RFBM AND NOT FORMAT;
TO_BE_INITIALIZED := (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        XINUSE[0] := 1;
        load x87 state from legacy region of XSAVE area;
    ELSIF TO_BE_INITIALIZED[0] = 1
        THEN
            XINUSE[0] := 0;
            initialize x87 state;
    FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        XINUSE[1] := 1;
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
    ELSIF TO_BE_INITIALIZED[1] = 1
        THEN
            set all XMM registers to 0;
            XINUSE[1] := 0;
            MXCSR := 1F80H;
    FI;

NEXT_FEATURE_OFFSET = 576; // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    XINUSE[i] := 1;
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + \( n\) (\( n\) enumerated by CPUID(EAX=0DH,ECX=i):EAX);
        FI;
    IF TO_BE_INITIALIZED[i] = 1
        THEN
            XINUSE[i] := 0;
            initialize XSAVE state component i;
    FI;
ENDFOR;

XMODIFIED := NOT RFBM;

IF in VMX non-root operation
THEN VMXNR := 1;
ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
XRSTOR_INFO := (CPL, VMXNR, LAXA, COMPMASK);

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent
XRSTORS: void _xrstors( void *, unsigned __int64);
XRSTORS64: void _xrstors64( void *, unsigned __int64);

Protected Mode Exceptions
#GP(0) 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 the LOCK prefix is used.

Real-Address Mode Exceptions
#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.
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 the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.
Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)  If 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 the LOCK prefix is used.
XSAVE—Save Processor Extended States

<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>NP 0F AE /4 XSAVE mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVE</td>
<td>Save state components specified by EDX:EAX to mem.</td>
</tr>
<tr>
<td>NP REX.W + 0F AE /4 XSAVE64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSAVE</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 (r, 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. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, "x87 State" 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" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) 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" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) 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" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

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. 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.
Operation

\[
\text{RFBM := XCR0 AND EDX:EAX; /* bitwise logical AND */}
\]

\[
\text{OLD_BV := XSTATE_BV field from XSAVE header;}
\]

IF RFBM[0] = 1
  THEN store x87 state into legacy region of XSAVE area;
  FI;

IF RFBM[1] = 1
  THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
  FI;

  THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
  FI;

FOR i := 2 TO 62
  IF RFBM[i] = 1
    THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
  ENDFOR;

\[
\text{XSTATE_BV field in XSAVE header := (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM)};
\]

Flags Affected
None.

Intel C/C++ Compiler Intrinsic Equivalent

\[
\text{XSAVE: \ void \_xsave( void *, unsigned \_int64);} \\
\text{XSAVE: \ void \_xsave64( void *, unsigned \_int64);} \\
\]

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
   If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM If CR0.TS[bit 3] = 1.
#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.
   If CR4.OSXSAVE[bit 18] = 0.
   If the LOCK prefix is used.

Virtual-8086 Mode Exceptions
Same exceptions as in protected mode.

Compatibility Mode Exceptions
Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) 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 the LOCK prefix is used.
#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory
   operand is not aligned on a 64-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).
INSTRUCTION SET REFERENCE, V-Z

XSAVEC—Save Processor Extended States with Compaction

<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>NP 0F C7 /4 XSAVEC mem</td>
<td>M</td>
<td>V/V</td>
<td>XSAVEC</td>
<td>Save state components specified by EDX:EAX to mem with compaction.</td>
</tr>
<tr>
<td>NP REX.W + 0F C7 /4 XSAVECG64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSAVEC</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*. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” 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 \( \text{RFBM}[i] = 1 \) 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” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.)
- 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” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*).
- 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” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.) 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” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*).

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, 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.
**Operation**

```
RFBM := XCR0 AND EDX:EAX; /* bitwise logical AND */
TO_BE_SAVED := RFBM AND XINUSE; /* bitwise logical AND */
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
FI;
IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;
IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;
NEXT_FEATURE_OFFSET = 576; // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                FI;
                NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
            FI;
        FI;
ENDFOR;
XSTATE_BV field in XSAVE header := TO_BE_SAVED;
XCOMP_BV field in XSAVE header := RFBM OR 80000000_00000000H;
```

**Flags Affected**

None.

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

XSAVEC: `void _xsavec( void * , unsigned __int64);`
XSAVEC64: `void _xsavec64( void * , unsigned __int64);`

**Protected Mode Exceptions**

- **#GP(0)**: If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- **#SS(0)**: 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.XSAVEC[bit 1] = 0.
  If CR4.OSXSAVE[bit 18] = 0.
  If the LOCK prefix is used.
#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0) If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).
XSAVEOPT—Save Processor Extended States Optimized

<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>NP 0F AE /6 XSAVEOPT mem</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>NP REX.W + 0F AE /6 XSAVEOPT64 mem</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 (r, 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. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, ”x87 State” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.

Section 13.9, ”Operation of XSAVEOPT,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component $i$ only if RFBM[$i$] = 1 and XINUSE[$i$] = 1.¹ (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”of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) Even if both bits are 1, XSAVEOPT may optimize and not save state component $i$ if (1) state component $i$ has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, ”Legacy Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, ”XSAVE Header” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1) and writes a modified value back to memory as follows. If RFBM[$i$] = 1, XSAVEOPT writes XSTATE_BV[$i$] with the value of XINUSE[$i$]. If RFBM[$i$] = 0, XSAVEOPT writes XSTATE_BV[$i$] with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, ”Extended Region of an XSAVE Area” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

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.

¹. There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if RFBM[1] = 1 or RFBM[2] = 1, regardless of the value of XINUSE.
See Section 13.6, "Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

**Operation**

\[
\begin{align*}
\text{RFBM} & := \text{XCR0 AND EDX:EAX}; \quad /* \text{bitwise logical AND} */ \\
\text{OLD_BV} & := \text{XSTATE_BV field from XSAVE header}; \\
\text{TO\_BE\_SAVED} & := \text{RFBM AND XINUSE}; \\
\text{IF} & \text{ in VMX non-root operation} \\
& \quad \text{THEN} \text{ VMXNR} := 1; \quad \text{ELSE} \text{ VMXNR} := 0; \\
\text{Fl;} \\
\text{LAXA} & := \text{linear address of XSAVE area}; \\
\text{IF} & \text{ XRSTOR\_INFO} = \langle \text{CPL, VMXNR, LAXA, 00000000\_00000000H} \rangle \\
& \quad \text{THEN} \text{ TO\_BE\_SAVED} := \text{TO\_BE\_SAVED AND XMODIFIED}; \\
\text{Fl;} \\
\text{IF} & \text{ TO\_BE\_SAVED}[0] = 1 \\
& \quad \text{THEN} \text{ store x87 state into legacy region of XSAVE area}; \\
\text{Fl;} \\
\text{IF} & \text{ TO\_BE\_SAVED}[1] \\
& \quad \text{THEN} \text{ store XMM registers into legacy region of XSAVE area}; // \text{this step does not save MXCSR or MXCSR\_MASK} \\
\text{Fl;} \\
\text{IF} & \text{ RFBM}[1] = 1 \text{ or RFBM}[2] = 1 \\
& \quad \text{THEN} \text{ store MXCSR and MXCSR\_MASK into legacy region of XSAVE area}; \\
\text{Fl;} \\
\text{FOR} & \text{ i := 2} \text{T}O \text{ 62} \\
& \quad \text{IF} \text{ TO\_BE\_SAVED}[i] = 1 \\
& \quad \quad \text{THEN} \text{ save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX)}; \\
& \quad \quad \text{Fl;} \\
\text{ENDFOR;} \\
\text{XSTATE_BV field in XSAVE header} & := (\text{OLD\_BV AND NOT RFBM}) \text{ OR (XINUSE AND RFBM)}; \\
\end{align*}
\]

**Flags Affected**

None.

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

XSAVEOPT: `void _xsaveopt(void *, unsigned __int64);`

XSAVEOPT: `void _xsaveopt64(void *, unsigned __int64);`

**Protected Mode Exceptions**

- **#GP(0)** If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
  - If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
- **#SS(0)** If a memory operand effective address is outside the SS segment limit.
- **#PF(fault-code)** If a page fault occurs.
- **#NM** If CR0.TS[bit 3] = 1.


#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
If any part of the operand lies outside the effective address space from 0 to FFFFH.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

#PF(fault-code) If a page fault occurs.

#NM If CR0.TS[bit 3] = 1.

#UD If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
If CR4.OSXSAVE[bit 18] = 0.
If the LOCK prefix is used.

#AC If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-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).
XSAVES—Save Processor Extended States Supervisor

<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>NP 0F C7 /5 XSAVES mem</td>
<td>M</td>
<td>V/V</td>
<td>XSS</td>
<td>Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.</td>
</tr>
<tr>
<td>NP REX.W + 0F C7 /5 XSAVES64 mem</td>
<td>M</td>
<td>V/N.E.</td>
<td>XSS</td>
<td>Save state components specified by EDX:EAX to mem with compaction, optimizing if possible.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
<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>
<td></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. Like FXRSTOR and FXSAVE, the memory format used for x87 state depends on a REX.W prefix; see Section 13.5.1, “x87 State” 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 \( \text{RFBM}[i] = 1 \) 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" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) Even if both bits are 1, XSAVES may optimize and not save state component \( i \) if (1) state component \( i \) has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).
- 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” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) 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” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1).

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.
See Section 13.6, "Processor Tracking of XSAVE-Managed State," of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

**Operation**

```plaintext
RFBM := (XCR0 OR IA32_XSS) AND EDX:EAX; /* bitwise logical OR and AND */
IF in VMX non-root operation
    THEN VMXNR := 1;
    ELSE VMXNR := 0;
FI;
LAXA := linear address of XSAVE area;
COMPMASK := RFBM OR 80000000_00000000H;
TO_BE_SAVED := RFBM AND XINUSE;
IF XRSTOR_INFO = (CPL,VMXNR,LAXA,COMPMASK)
    THEN TO_BE_SAVED := TO_BE_SAVED AND XMODIFIED;
FI;
IF MXCSR ≠ 1F80H AND RFBM[1]
    THEN TO_BE_SAVED[1] = 1;
FI;
IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;
IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;
NEXT_FEATURE_OFFSET = 576; // Legacy area and XSAVE header consume 576 bytes
FOR i := 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN
                    save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    IF i = 8 // state component 8 is for PT state
                        THEN IA32_RTIT_CTL.TraceEn[bit 0] := 0;
                    FI;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
        FI;
ENDFOR;
NEW_HEADER := RFBM AND XINUSE;
IF MXCSR ≠ 1F80H AND RFBM[1]
    THEN NEW_HEADER[1] = 1;
FI;
XSTATE_BV field in XSAVE header := NEW_HEADER;
XCOMP_BV field in XSAVE header := COMPMASK;
```

**Flags Affected**

None.
Intel C/C++ Compiler Intrinsic Equivalent

XSAVES:    void _xsaves( void *, unsigned __int64);
XSAVES64:  void _xsaves64( void *, unsigned __int64);

Protected Mode Exceptions

#GP(0)     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.
#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 the LOCK prefix is used.

Real-Address Mode Exceptions

#GP        If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
           If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM        If CR0.TS[bit 3] = 1.
#UD        If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
           If CR4.OSXSAVE[bit 18] = 0.
           If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)     If CPL > 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 the LOCK prefix is used.
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

XCR[ECX] := 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 26] = 0.
  - If the LOCK 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.

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>NP OF 01 D6 XTEST</td>
<td>ZO</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>Operand1</th>
<th>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</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

XTEST

IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF := 0
    ELSE
        ZF := 1
FI;

### Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF flags are cleared.

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

XTEST: 

```
int _xtest( void );
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

#UD 

CPUID.(EAX=7, ECX=0):EBX.HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.

If LOCK prefix is used.