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

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

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

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

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

<table>
<thead>
<tr>
<th>Imm</th>
<th>[7:4]</th>
<th>[3:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>FALSE</td>
<td>andAnorBC norBnandAC andAIB norCnandBA andAIC andAxorBC andA?nandBC</td>
</tr>
<tr>
<td>04H</td>
<td>andBnorAC</td>
<td>norCxnorBA B?norACandAC B?norACA andBIC norCnoraBA B?ICandAC B?ICA</td>
</tr>
<tr>
<td>06H</td>
<td>norAxnorBC</td>
<td>A?norBC:xorBC B?norAC:C xorBorAC C?norBA:B xorCorBA xorCB B?ICorAC</td>
</tr>
<tr>
<td>0CH</td>
<td>andBIA</td>
<td>A?norBxCB B?IA:andAC xorBA andBnandAC A?IC:B xorBandAC A?nandBC:B</td>
</tr>
<tr>
<td>0FH</td>
<td>IA</td>
<td>nandAorBC C?nandBA:IA nandBA B?nandAC:IA nandCA nandAxBnorBC nandABC</td>
</tr>
</tbody>
</table>

Table 5-2 shows the half of 256-entry map corresponding to column index values 8:15.
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 VTERNLOG syntax. There is a unique set of three byte constants (F0H, CCH, AAH) that can be used for this purpose as input operands in conjunction with the Boolean expressions defined in those tables. The reverse mapping can be expressed as:

\[
\text{Result}_{\text{imm8}} = \text{Table\_Lookup\_Entry}(0F0H, 0CCH, 0AAH)
\]

\(\text{Table\_Lookup\_Entry}\) is the Boolean expression defined in Table 5-1 and Table 5-2.

<table>
<thead>
<tr>
<th>Imm</th>
<th>[3:0]</th>
<th>08H</th>
<th>09H</th>
<th>0AH</th>
<th>0BH</th>
<th>0CH</th>
<th>0DH</th>
<th>0EH</th>
<th>0FH</th>
</tr>
</thead>
<tbody>
<tr>
<td>00H</td>
<td>andABC</td>
<td>andA\text{xnor}BC</td>
<td>andCA</td>
<td>B?andAC:A</td>
<td>andBA</td>
<td>C?andBA:A</td>
<td>andAorBC</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>01H</td>
<td>A?andBC\text{nor}BC</td>
<td>B?andAC:C</td>
<td>C7B:oB</td>
<td>A?B:orBC</td>
<td>B?A:C</td>
<td>xnorAorBC</td>
<td>orAnorBC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03H</td>
<td>A?andBC!B</td>
<td>xnorB\text{n}andAC</td>
<td>A7C:oB</td>
<td>nandB\text{n}andAC</td>
<td>xnorBA</td>
<td>B?A:nandAC</td>
<td>A?orBC!B</td>
<td>orAlB</td>
<td></td>
</tr>
<tr>
<td>05H</td>
<td>A?andBC:IC</td>
<td>xnor\text{C}andBA</td>
<td>xnorCA</td>
<td>C7\text{an}d\text{C}andBA</td>
<td>A?B:IC</td>
<td>nand\text{C}andBA</td>
<td>A?orBC:IC</td>
<td>orAlC</td>
<td></td>
</tr>
<tr>
<td>07H</td>
<td>xnorA\text{a}ndBC</td>
<td>A?xnorBC:andBC</td>
<td>C7\text{andBA:ndBC}</td>
<td>C7nnorC\text{a}ndBA</td>
<td>nandBxorAC</td>
<td>A?B:nandBC</td>
<td>nandCxorBA</td>
<td>A?orBC:nandB \ C</td>
<td>orAn\text{and}BC</td>
</tr>
<tr>
<td>08H</td>
<td>andCB</td>
<td>A?xnorBC:andBC</td>
<td>andCorAB</td>
<td>B?CA</td>
<td>andBorAC</td>
<td>C?B:A</td>
<td>majorABC</td>
<td>orA\text{and}BC</td>
<td></td>
</tr>
<tr>
<td>09H</td>
<td>B?C\text{no}rAC</td>
<td>xnorCB</td>
<td>xnorC\text{orr}BA</td>
<td>B?orBA:IB</td>
<td>xnorBorAC</td>
<td>B?orAC:C</td>
<td>orA\text{or}BCxnorB \ C</td>
<td>orA\text{nx}norBC</td>
<td></td>
</tr>
<tr>
<td>0AH</td>
<td>A?andBC\text{C}</td>
<td>A?xnorBC \ C</td>
<td>C</td>
<td>B?CorAC</td>
<td>A?B:CA</td>
<td>B?orAC:xorAC</td>
<td>or\text{C}andBA</td>
<td>orCA</td>
<td></td>
</tr>
<tr>
<td>0BH</td>
<td>B?C\text{IA}</td>
<td>B?C\text{nda}ndAC</td>
<td>orC\text{n}orBA</td>
<td>orCB</td>
<td>B?orAC:IA</td>
<td>B?orAC:nandAC</td>
<td>orC\text{nxorBA}</td>
<td>nandB\text{or}AC</td>
<td></td>
</tr>
<tr>
<td>0CH</td>
<td>A?andBCB</td>
<td>A?xnorBC:B</td>
<td>A?CB</td>
<td>C7\text{orBA:xorBA}</td>
<td>B</td>
<td>C?B:orBA</td>
<td>or\text{Band}AC</td>
<td>orBA</td>
<td></td>
</tr>
<tr>
<td>0DH</td>
<td>C?BIA</td>
<td>C?B:nnor\text{BA}</td>
<td>C7\text{orBA:IA}</td>
<td>C7\text{orBA:nandBA}</td>
<td>orB\text{nor}AC</td>
<td>or\text{B}IC</td>
<td>orB\text{n}xorAC</td>
<td>nandC\text{nor}BA</td>
<td></td>
</tr>
<tr>
<td>0EH</td>
<td>A?andBC\text{orBC}</td>
<td>A?xnorBC\text{orBC}</td>
<td>A?C\text{orBC}</td>
<td>orC\text{xorBA}</td>
<td>A?B:orBC</td>
<td>orB\text{x}orAC</td>
<td>orCB</td>
<td>orABC</td>
<td></td>
</tr>
<tr>
<td>0FH</td>
<td>nandA\text{nn}orBC</td>
<td>nandA\text{xorBC}</td>
<td>orCIA</td>
<td>orC\text{n}andBA</td>
<td>or\text{B}IA</td>
<td>orB\text{rn}andAC</td>
<td>nandAnorBC</td>
<td>TRUE</td>
<td></td>
</tr>
</tbody>
</table>
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>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

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

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

**VALIGND (EVEX encoded versions)**

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

IF (SRC2 *is memory*) (AND EVEX.b = 1)
THEN
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
  \[\]
VALIGNQ (EVEX encoded versions)

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

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

tmp[VL-1:0] := src[VL-1:0]
tmp[2VL-1:VL] := SRC1[VL-1:0]
; Shift right quadword elements

IF VL = 128
  THEN SHIFT = imm8[0]
ELSE
  IF VL = 256
    THEN SHIFT = imm8[1:0]
  ELSE SHIFT = imm8[2:0]
  FI
FI;
tmp[2VL-1:0] := tmp[2VL-1:0] >> (64*SHIFT)
; Apply writemask

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

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</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend double-precision vector xmm2 and double-precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 65 /r VBLENDMPD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend double-precision vector ymm2 and double-precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 65 /r VBLENDMPD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend double-precision vector zmm2 and double-precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 65 /r VBLENDMPS xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend single-precision vector xmm2 and single-precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 65 /r VBLENDMPS ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Blend single-precision vector ymm2 and single-precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 65 /r VBLENDMPS zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Blend single-precision vector zmm2 and single-precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.</td>
</tr>
</tbody>
</table>

**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 \(KL-1\)

\(i := j \times 64\)

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

THEN

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

THEN

\(\text{DEST}[i+63:i] := \text{SRC2}[63:0]\)

ELSE

\(\text{DEST}[i+63:i] := \text{SRC2}[i+63:i]\)

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN \(\text{DEST}[i+63:i] := \text{SRC1}[i+63:i]\)

ELSE ; zeroing-masking

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

FI;

FI;

ENDFOR

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

**VBLENDMPS (EVEX encoded versions)**

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

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

\(i := j \times 32\)

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

THEN

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

THEN

\(\text{DEST}[i+31:i] := \text{SRC2}[31:0]\)

ELSE

\(\text{DEST}[i+31:i] := \text{SRC2}[i+31:i]\)

FI;

ELSE

IF *merging-masking* ; merging-masking

THEN \(\text{DEST}[i+31:i] := \text{SRC1}[i+31:i]\)

ELSE ; zeroing-masking

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

FI;

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL}-1:VL] := 0\)
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".
### VBROADCAST—Load with Broadcast Floating-Point Data

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32</td>
<td>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 VBROADCASTSS 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 VBROADCASTSD 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 VBROADCASTF128 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 VBROADCAST SS xmm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</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 VBROADCASTSS ymm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast low single-precision floating-point element in the source operand to eight locations in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Broadcast low double-precision floating-point element in the source operand to four locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 19 /r VBROADCASTSD ymm1 (k1)[z], xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast low double-precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 19 /r VBROADCASTSD zmm1 (k1)[z], xmm2/m64</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast low double-precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 19 /r VBROADCASTF32X2 ymm1 (k1)[z], xmm2/m64</td>
<td>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 VBROADCASTF32X2 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 VBROADCASTSS 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 VBROADCASTSS 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 VBROADCASTSS 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 VBROADCASTF32X4 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 VBROADCASTF32X4 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 VBROADCASTF64X2 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

**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**

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.

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

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

If 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.
Operation

**VBROADCASTSS (128 bit version VEX and legacy)**

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

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

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

**VBROADCASTSS (EVEX encoded versions)**

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

For \( j = 0 \) TO \( KL-1 \)

\[
i := j \times 32 \\
\text{IF} k1[j] \text{ OR } *\text{no writemask}* \\
\text{THEN} \text{DEST}[i+31:i] := \text{SRC}[31:0] \\
\text{ELSE} \\
\quad \text{IF} *\text{merging-masking}* \\
\quad \text{THEN} *\text{DEST}[i+31:i] remains unchanged* \\
\quad \text{ELSE} \\
\quad \quad \text{DEST}[i+31:i] := 0 \\
\quad \text{FI} \\
\text{FI;}
\]

EndFor

\[
\text{DEST}[:256] := 0
\]
VBROADCASTSD (VEX.256 encoded version)

\[ \text{temp} := \text{SRC}[63:0] \]
\[ \text{DEST}[63:0] := \text{temp} \]
\[ \text{DEST}[127:64] := \text{temp} \]
\[ \text{DEST}[191:128] := \text{temp} \]
\[ \text{DEST}[255:192] := \text{temp} \]
\[ \text{DEST}[	ext{MAXVL}-1:256] := 0 \]

VBROADCASTSD (EVEX encoded versions)

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

\[
\text{FOR } j := 0 \text{ TO KL-1} \]
\[
i := j \times 64 \]
\[
\text{IF } k1[j] \text{ OR } \text{no writemask} \]
\[
\text{THEN } \text{DEST}[i+63:i] := \text{SRC}[63:0] \]
\[
\text{ELSE} \]
\[
\text{IF } \text{merging-masking} \]
\[
\text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged} \]
\[
\text{ELSE} \]
\[
\text{DEST}[i+63:i] := 0 \]
\[
\text{FI} \]
\[
\text{FI} ; \]
\[
\text{ENDFOR} \]
\[ \text{DEST}[	ext{MAXVL}-1:VL] := 0 \]

VBROADCASTF32x2 (EVEX encoded versions)

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

\[
\text{FOR } j := 0 \text{ TO KL-1} \]
\[
i := j \times 32 \]
\[
n := (j \mod 2) \times 32 \]
\[
\text{IF } k1[j] \text{ OR } \text{no writemask} \]
\[
\text{THEN } \text{DEST}[i+31:i] := \text{SRC}[n+31:n] \]
\[
\text{ELSE} \]
\[
\text{IF } \text{merging-masking} \]
\[
\text{THEN } \text{DEST}[i+31:i] \text{ remains unchanged} \]
\[
\text{ELSE} \]
\[
\text{DEST}[i+31:i] := 0 \]
\[
\text{FI} \]
\[
\text{FI} ; \]
\[
\text{ENDFOR} \]
\[ \text{DEST}[	ext{MAXVL}-1:VL] := 0 \]

VBROADCASTF128 (VEX.256 encoded version)

\[ \text{temp} := \text{SRC}[127:0] \]
\[ \text{DEST}[127:0] := \text{temp} \]
\[ \text{DEST}[255:128] := \text{temp} \]
\[ \text{DEST}[	ext{MAXVL}-1:256] := 0 \]
**VBROADCASTF32X4 (EVEX encoded versions)**

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

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

\[i := j \times 32\]

\[n := (j \mod 4) \times 32\]

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

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

ELSE

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

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

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

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

FI

FI;

ENDFOR

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

**VBROADCASTF64X2 (EVEX encoded versions)**

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

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

\[i := j \times 64\]

\[n := (j \mod 2) \times 64\]

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

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

ELSE

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

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

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

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

FI

FI;

ENDFOR;

**VBROADCASTF32X8 (EVEX.U1.512 encoded version)**

FOR \(j := 0\) TO 15

\[i := j \times 32\]

\[n := (j \mod 8) \times 32\]

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

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

ELSE

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

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

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

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

FI

FI;

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)
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 __m512_bROADCAST_f32x2(__m128 a);
VBROADCASTF32x2 __m512 __m512_mask_broadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 __m512_maskz_broadcast_f32x2(__mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 __m512_bROADCAST_f32x2(__m128 a);
VBROADCASTF32x4 __m512 __m512_bROADCAST_f32x4(__m128 a);
VBROADCASTF32x4 __m512 __m512_mask_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 __m512_maskz_broadcast_f32x4(__mmask16 k, __m128 a);
VBROADCASTF64x2 __m512d __m512d_bROADCAST_f64x2(__m128d a);
VBROADCASTF64x2 __m512d __m512d_mask_broadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d __m512d_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d __m512d_bROADCAST_f64x2(__m128d a);
VBROADCASTF64x4 __m512d __m512d_bROADCAST_f64x4(__m512d a);
VBROADCASTF64x4 __m512d __m512d_mask_broadcast_f64x4(__m512d s, __mmask8 k, __m512d a);
VBROADCASTF64x4 __m512d __m512d_maskz_broadcast_f64x4(__mmask8 k, __m512d a);
VBROADCASTF64x4 __m512d __m512d_bROADCAST_f64x4(__m512d a);
VBROADCASTSD __m512d __m512d_broadcastsd_pd( double *a);
VBROADCASTSD __m512d __m512d_mask_broadcastsd_pd( double *a);
VBROADCASTSS __m512 __m512_bROADCASTss_ps(__m128 a);
VBROADCASTSS __m512 __m512_mask_broadcastss_ps(__m128 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512 __m512_maskz_broadcastss_ps(__mmask16 k, __m128 a);
VBROADCASTSS __m512 __m512_bROADCASTss_ps(__m128 a);
VBROADCASTSS __m512 __m512_mask_broadcastss_ps(__m512 s, __mmask8 k, __m128 a);
VBROADCASTSS __m512 __m512_maskz_broadcastss_ps(__mmask8 k, __m128 a);
VBROADCASTSS __m512 __m512_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

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.w1 8A /r VCOMPRESSPD xmm1/m128 {k1}{z}, xmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed double-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 8A /r VCOMPRESSPD ymm1/m256 {k1}{z}, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Compress packed double-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w1 8A /r VCOMPRESSPD zmm1/m512 {k1}{z}, zmm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Compress packed double-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.</td>
</tr>
</tbody>
</table>

**Description**

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

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

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

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

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

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

**Operation**

VCOMPRESSPD (EVEX encoded versions) store form

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

SIZE := 64
k := 0
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      DEST[k+SIZE-1:k] := SRC[i+63:i]
      k := k + SIZE
  FI;
ENDFOR
VCOMPRESSPD (EVEX encoded versions) reg-reg form

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

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

Intel C/C++ Compiler Intrinsic Equivalent
VCOMPRESSPD __m512d _mm512_mask_compress_pd( __m512d s, __mmask8 k, __m512d a);
VCOMPRESSPD __m512d _mm512_maskz_compress_pd( __mmask8 k, __m512d a);
VCOMPRESSPD void _mm512_mask_compressstoreu_pd( void * d, __mmask8 k, __m512d a);
VCOMPRESSPD __m256d _mm256_mask_compress_pd( __m256d s, __mmask8 k, __m256d a);
VCOMPRESSPD __m256d _mm256_maskz_compress_pd( __mmask8 k, __m256d a);
VCOMPRESSPD void _mm256_mask_compressstoreu_pd( void * d, __mmask8 k, __m256d a);
VCOMPRESSPD __m128d _mm_mask_compress_pd( __m128d s, __mmask8 k, __m128d a);
VCOMPRESSPD __m128d _mm_maskz_compress_pd( __mmask8 k, __m128d a);
VCOMPRESSPD void _mm_mask_compressstoreu_pd( void * d, __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instructions, see Exceptions Type E4.nb 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>CPUD 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>

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

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

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

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

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

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

**Operation**

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

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

\[\text{SIZE} := 32\]

\[k := 0\]

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

\[i := j \ast 32\]

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

\[\text{THEN}\]

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

\[k := k + \text{SIZE}\]

\[\text{FI}\]

\[\text{ENDFOR}\]
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 _mm512_mask_compress_ps( __m512 s, __mmask16 k, __m512 a);
VCOMPRESSPS __m512 _mm512_maskz_compress_ps( __mmask16 k, __m512 a);
VCOMPRESSPS void _mm512_mask_compressstoreu_ps( void * d, __mmask16 k, __m512 a);
VCOMPRESSPS __m256 _mm256_mask_compress_ps( __m256 s, __mmask8 k, __m256 a);
VCOMPRESSPS __m256 _mm256_maskz_compress_ps( __mmask8 k, __m256 a);
VCOMPRESSPS void _mm256_mask_compressstoreu_ps( void * d, __mmask8 k, __m256 a);
VCOMPRESSPS __m128 _mm_mask_compress_ps( __m128 s, __mmask8 k, __m128 a);
VCOMPRESSPS __m128 _mm_maskz_compress_ps( __mmask8 k, __m128 a);
VCOMPRESSPS void _mm_mask_compressstoreu_ps( void * d, __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

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

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<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.F2.0F38.W0 72 /r VCVTNE2PS2BF16 xmm1(k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Convert packed single data from xmm2 and xmm3/m128/m32bcst to packed BF16 data in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F38.W0 72 /r VCVTNE2PS2BF16 ymm1(k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512_BF16</td>
<td>Convert packed single data from ymm2 and ymm3/m256/m32bcst to packed BF16 data in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F38.W0 72 /r VCVTNE2PS2BF16 zmm1(k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F AVX512_BF16</td>
<td>Convert packed single data from zmm2 and zmm3/m512/m32bcst to packed BF16 data 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>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**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
VL = (128, 256, 512)
KL = VL/16

origdest := dest
FOR i := 0 to KL-1:
  IF k1[i] or *no writemask*:
    IF i < KL/2:
      IF src2 is memory and evex.b == 1:
        t := src2.fp32[0]
      ELSE:
        t := src2.fp32[i]
    ELSE:
      t := src1.fp32[i-KL/2]
    // See VCVTNEPS2BF16 for definition of convert helper function
    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] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VCVTNE2PS2BF16 __m128bh _mm_cvtne2ps_pbh (__m128bh, __m128);
VCVTNE2PS2BF16 __m128bh _mm_mask_cvtne2ps_pbh (__m128bh, __mmask8, __m128bh, __m128);
VCVTNE2PS2BF16 __m128bh _mm_maskz_cvtne2ps_pbh (__mmask8, __m128bh, __m128);
VCVTNE2PS2BF16 __m256bh _mm256_cvtne2ps_pbh (__m256bh, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_mask_cvtne2ps_pbh (__m256bh, __mmask16, __m256bh, __m256);
VCVTNE2PS2BF16 __m256bh _mm256_maskz_cvtne2ps_pbh (__mmask16, __m256bh, __m256);
VCVTNE2PS2BF16 __m512bh _mm512_cvtne2ps_pbh (__m512bh, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_mask_cvtne2ps_pbh (__m512bh, __mmask32, __m512bh, __m512);
VCVTNE2PS2BF16 __m512bh _mm512_maskz_cvtne2ps_pbh (__mmask32, __m512bh, __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 Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| EVEX.128.F3.0F38.W0 72 /r  
VCVTNEPS2BF16 xmm1{k1}{z}, xmm2/m128/m32bcst | A      | V/V                   | AVX512VL          | Convert packed single data from xmm2/m128 to packed BF16 data in xmm1 with writemask k1. |
| EVEX.256.F3.0F38.W0 72 /r  
VCVTNEPS2BF16 xmm1{k1}{z}, ymm2/m256/m32bcst | A      | V/V                   | AVX512VL          | Convert packed single data from ymm2/m256 to packed BF16 data in xmm1 with writemask k1. |
| EVEX.512.F3.0F38.W0 72 /r  
VCVTNEPS2BF16 ymm1{k1}{z}, zmm2/m512/m32bcst | A      | V/V                   | AVX512F           | Convert packed single data from zmm2/m512 to packed BF16 data in ymm1 with writemask k1. |

## 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>ModRMr/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[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 (__m256bh, __mmask16, __m256);
VCVTNEPS2BF16 __m128bh _mm256_maskz_cvtneps_pbh (__mmask16, __m256);
VCVTNEPS2BF16 __m256bh _mm512_cvtneps_pbh (__m512);
VCVTNEPS2BF16 __m256bh _mm512_mask_cvtneps_pbh (__m512bh, __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”.
VCVTD2QQ—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 VCVTD2QQ 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/m128/m64bcst to two packed quadword integers in xmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 7B /r VCVTD2QQ 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 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 7B /r VCVTD2QQ 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/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1.</td>
<td></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* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
         ELSE ; zeroing-masking
            DEST[i+63:i] := 0
         FI
   FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

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

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

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

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

VCVTPD2QQ __m512i __mm512_cvtpd_epi64( __m512d a);
VCVTPD2QQ __m512i __mm512_mask_cvtpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i __mm512_maskz_cvtpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m512i __mm512_cvt_roundpd_epi64( __m512d a, int r);
VCVTPD2QQ __m512i __mm512_mask_cvt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m512i __mm512_maskz_cvt_roundpd_epi64( __mmask8 k, __m512d a);
VCVTPD2QQ __m256i __mm256_cvtpd_epi64( __m256d a);
VCVTPD2QQ __m128i __mm128_cvtpd_epi64( __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.
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 Flags</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

ENDFOR

DEST[MAXVL-1:VL/2] := 0

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

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

FOR j := 0 TO KL-1

    i := j * 32

    k := j * 64

    IF k1[j] OR *no writemask*

    THEN

        IF (EVEX.b = 1)

            THEN

                DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0])

            ELSE

                DEST[i+31:i] := Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])

            FI;

        ELSE

            IF *merging-masking* ; merging-masking

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

            ELSE ; zeroing-masking

                DEST[i+31:i] := 0

            FI

        FI

ENDFOR

DEST[MAXVL-1:VL/2] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VCVTPD2UDQ __m256i _mm512_cvtpd_epu32( __m512d a);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_maskz_cvtpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_cvt_roundpd_epu32( __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_mask_cvt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_maskz_cvt_roundpd_epu32( __mmask8 k, __m512d a);
VCVTPD2UDQ __m128i _mm256_mask_cvtpd_epu32( __m128i s, __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 79 /r</td>
<td>A/V/V</td>
<td>AVX512VL AVX512DQ</td>
<td></td>
<td>Convert two packed double-precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W1 79 /r</td>
<td>A/V/V</td>
<td>AVX512VL AVX512DQ</td>
<td></td>
<td>Convert fourth packed double-precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W1 79 /r</td>
<td>A/V/V</td>
<td>AVX512DQ</td>
<td></td>
<td>Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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
    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[i+63:i])
                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
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI
ENDFOR

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

VCVTPD2UQQ __m512i __mm512_cvtpd_epu64(__m512d a);
VCVTPD2UQQ __m512i __mm512_mask_cvtpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i __mm512_maskz_cvtpd_epu64(__mmask8 k, __m512d a);
VCVTPD2UQQ __m512i __mm512_cvt_roundpd_epu64(__m512d a, int r);
VCVTPD2UQQ __m512i __mm512_mask_cvt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i __mm512_maskz_cvt_roundpd_epu64(__mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i __mm256_cvt_roundpd_epu64(__m256d a);
VCVTPD2UQQ __m128i __mm128_cvt_roundpd_epu64(__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.
VCVTPH2PS—Convert 16-bit FP values to Single-Precision FP values

### 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>VEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64</td>
<td>A V/V</td>
<td>F16C</td>
<td>Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128</td>
<td>A V/V</td>
<td>F16C</td>
<td>Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1 {k1}[z], xmm2/m64</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point values in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1 {k1}[z], xmm2/m128</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point values in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 13 /r VCVTPH2PS zmm1 {k1}[z], ymm2/m256 (sae)</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Convert sixteen packed half precision (16-bit) floating-point values in ymm2/m256 to packed single-precision floating-point values in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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**

```c
vCvt_h2s(SRC1[15:0])
{
    RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}
```

**VCVTPH2PS (EVEX encoded versions)**

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

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := vCvt_h2s(SRC[k+15:k])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI
ENDFOR
DEST[MAXVL-1:VL] := 0

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

DEST[31:0] := vCvt_h2s(SRC1[15:0]);
DEST[63:32] := vCvt_h2s(SRC1[31:16]);
DEST[95:64] := vCvt_h2s(SRC1[47:32]);
DEST[127:96] := vCvt_h2s(SRC1[63:48]);
DEST[159:128] := vCvt_h2s(SRC1[79:64]);
DEST[191:160] := vCvt_h2s(SRC1[95:80]);
DEST[223:192] := vCvt_h2s(SRC1[111:96]);
DEST[255:224] := vCvt_h2s(SRC1[127:112]);
DEST[MAXVL-1:256] := 0
VCVTPH2PS (VEAX.128 encoded version)
DEST[31:0] := vCvt_h2s(SRC1[15:0]);
DEST[63:32] := vCvt_h2s(SRC1[31:16]);
DEST[95:64] := vCvt_h2s(SRC1[47:32]);
DEST[127:96] := vCvt_h2s(SRC1[63:48]);
DEST[MAXVL-1:128] := 0

Flags Affected
None

Intel C/C++ Compiler Intrinsic Equivalent
VCVTPH2PS __m512__mm512_cvtph_ps(__m256i a);
VCVTPH2PS __m512__mm512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);
VCVTPH2PS __m512__mm512_maskz_cvtph_ps(__mmask16 k, __m256i a);
VCVTPH2PS __m512__mm512_cvt_roundph_ps( __m256i a, int sae);
VCVTPH2PS __m512__mm512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m512__mm512_maskz_cvt_roundph_ps( __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m256__mm256_cvtph_ps(__m256 s, __mmask8 k, __m128i a);
VCVTPH2PS __m256__mm256_mask_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m256__mm256_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m256__mm256_cvt_roundph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m256__mm256_mask_cvt_roundph_ps(__mmask8 k, __m128i a, int sae);
VCVTPH2PS __m256__mm256_maskz_cvt_roundph_ps(__mmask8 k, __m128i a, int sae);

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

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F3A.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{sae}, 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.

![Figure 5-7. VCVTPS2PH (128-bit Version)](image)

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.


VCVT_S2H(SRC[31:0])
[
    IF Imm[2] = 0
    THEN ; using Imm[1:0] for rounding, see Table 5-3
        RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC[31:0]);
    ELSE ; using MXCSR.RC for rounding
        RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC[31:0]);
    FI;
]

VCVTPS2PH (EVEX encoded versions) when dest is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
    i := j * 16
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] :=
            vCvt_s2h(SRC[k+31:k])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+15:i] := 0
            FI
        FI
    FI;
ENDFOR
DEST[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>
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
  FIN;
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_cvtps_ph(__m256i s, __mmask16 k, __m512 a, const int imm);
VCVTPS2PH __m256i _mm512_maskz_cvtps_ph(__mmask16 k, __m512 a, const int imm);
VCVTPS2PH __m256i _mm256_cvtps_ph(__m256 m1, const int imm);
VCVTPS2PH __m256i _mm256_mask_cvtps_ph(__mmask8 k, __m256 a);
VCVTPS2PH __m256i _mm256_maskz_cvtps_ph(__mmask8 k, __m256 a);
VCVTPS2PH __m256i _mm256_cvt_roundps_ph(__m256 s, __mmask8 k, __m256 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/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>
<td></td>
</tr>
<tr>
<td>EVEX.256.0F.W0 79 /r VCVTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A/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>
<td></td>
</tr>
<tr>
<td>EVEX.512.0F.W0 79 /r VCVTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}</td>
<td>A/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>
<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 sixteen packed single-precision floating-point values in the source operand to sixteen unsigned double-word 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;
 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;
        ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2UDQ __m512i __m512_cvt_roundps_epu32( __m512 a);
VCVTPS2UDQ __m512i __mm512_cvt_roundps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i __mm512_cvt_roundps_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_mask_cvt_roundps_epu32( __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i __mm256_cvt_roundps_epu32( __m256d a);
VCVTPS2UDQ __m256i __mm256_mask_cvt_roundps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i __mm256_mask_cvt_roundps_epu32( __mmask8 k, __m256 a);
VCVTPS2UDQ __m128i __mm128_cvt_roundps_epu32( __m128d a);
VCVTPS2UDQ __m128i __mm128_mask_cvt_roundps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i __mm128_mask_cvt_roundps_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

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

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

\(\text{THEN}\)

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

\(\text{ELSE}\)

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

\(\text{FI};\)

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

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

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

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

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

\(\text{ELSE}\)

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

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

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

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

\(\text{FI}\)

\(\text{FI};\)

\(\text{ENDFOR}\)

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

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

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

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

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

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

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

\(\text{THEN}\)

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

\(\text{THEN}\)

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

\(\text{ELSE}\)

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

\(\text{FI}\)

\(\text{ELSE}\)

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

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

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

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

\(\text{FI}\)

\(\text{FI}\)

\(\text{ENDFOR}\)

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

VCVTPS2QQ __m512i __mm512_cvtps_epi64( __m512 a);
VCVTPS2QQ __m512i __mm512_mask_cvtps_epi64( __m512 s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i __mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTPS2QQ __m512i __mm512_cvtrndps_epi64( __m512 a, int r);
VCVTPS2QQ __m512i __mm512_mask_cvtrndps_epi64( __m512 s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i __mm512_maskz_cvtrndps_epi64( __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i __mm256_cvtps_epi64( __m256 a);
VCVTPS2QQ __m256i __mm256_mask_cvtps_epi64( __m256 s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i __mm256_maskz_cvtps_epi64( __mmask8 k, __m256 a);
VCVTPS2QQ __m128i __mm128_cvtps_epi64( __m128 a);
VCVTPS2QQ __m128i __mm128_mask_cvtps_epi64( __m128 s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i __mm128_maskz_cvtps_epi64( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see 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</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 79 /r VCVTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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)

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_UQuadInteger(SRC[k+31:k])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI
    FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

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

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

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

ENDFOR

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

VCVTPS2UQQ __m512i_mm512_cvtps_epu64(__m512 a);
VCVTPS2UQQ __m512i_mm512_mask_cvtps_epu64(__m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i_mm512_maskz_cvtps_epu64(__mmask16 k, __m512 a);
VCVTPS2UQQ __m512i_mm512_cvt_roundps_epu64(__m512 a, int r);
VCVTPS2UQQ __m512i_mm512_mask_cvt_roundps_epu64(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i_mm512_maskz_cvt_roundps_epu64(__mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i_mm256_cvtps_epu64(__m256 a);
VCVTPS2UQQ __m256i_mm256_mask_cvtps_epu64(__m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i_mm256_maskz_cvtps_epu64(__mmask8 k, __m256 a);
VCVTPS2UQQ __m128i_mm128_cvtps_epu64(__m128 a);
VCVTPS2UQQ __m128i_mm128_mask_cvtps_epu64(__m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i_mm128_maskz_cvtps_epu64(__mmask8 k, __m128 a);

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see 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
   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_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
   ELSE
      IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE  ; zeroing-masking
         DEST[i+63:i] := 0
      FI
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VCVTQQ2PD (EVEX encoded versions) when src operand is a memory source

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

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

DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PD __m512d _mm512_cvtepi64_pd( __m512i a);
VCVTQQ2PD __m512d _mm512_mask_cvtepi64_pd( __m512d s, __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_maskz_cvtepi64_pd( __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_cvt_roundepi64_pd( __m512i a, int r);
VCVTQQ2PD __m512d _mm512_mask_cvt_roundepi64_pd( __m512d s, __mmask8 k, __m512i a, int r);
VCVTQQ2PD __m512d _mm512_maskz_cvt_roundepi64_pd( __mmask8 k, __m512i a);
VCVTQQ2PD __m256d _mm256_mask_cvtepi64_pd( __m256d s, __mmask8 k, __m256i a);
VCVTQQ2PD __m256d _mm256_maskz_cvtepi64_pd( __mmask8 k, __m256i a);
VCVTQQ2PD __m128d _mm_mask_cvtepi64_pd( __m128d s, __mmask8 k, __m128i a);
VCVTQQ2PD __m128d _mm_maskz_cvtepi64_pd( __mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see 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>

**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 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 _mm256_cvtepi64_ps( __m512i a);
VCVTQQ2PS __m256 _mm256_mask_cvtepi64_ps( __m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm256_maskz_cvtepi64_ps( __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm256_cvtepi64_ps( __m512i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVE-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</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</td>
<td>A</td>
<td>V/N.E.¹</td>
<td>AVX512F</td>
<td>Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.</td>
</tr>
</tbody>
</table>

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

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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_cvt_roundsd_u32(__m128d, int r);
VCVTSD2USI unsigned __int64 _mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned __int64 _mm_cvt_roundsd_u64(__m128d, int r);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-48, "Type E3NF Class Exception Conditions".

Vol. 2C 5-57
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 (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 _mm_cvtss_u32(__m128 a);
VCVTSS2USI unsigned _mm_cvt_roundss_u32(__m128 a, int r);
VCVTSS2USI unsigned __int64 _mm_cvtss_u64(__m128 a);
VCVTSS2USI unsigned __int64 _mm_cvt_roundss_u64(__m128 a, int r);
SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions”.

VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer
VCVTTDP2QQ—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 VCVTTDP2QQ 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 VCVTTDP2QQ 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 VCVTTDP2QQ 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

**VCVTTDP2QQ (EVEX encoded version) when src operand is a register**

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

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

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

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

\[
\text{FOR } j := 0 \text{ TO } KL-1 \\
i := j \times 64 \\
\text{IF } k1[j] \text{ OR } \text{no writemask} \\
\text{THEN} \\
\text{IF } (\text{EVEX.b} == 1) \\
\text{THEN} \\
\text{DEST}[i+63:i] := \text{Convert Double-Precision Floating Point To QuadInteger_Truncate}(\text{SRC}[63:0]) \\
\text{ELSE} \\
\text{DEST}[i+63:i] := \text{Convert Double-Precision Floating Point To QuadInteger_Truncate}(\text{SRC}[i+63:i]) \\
\text{FI;} \\
\text{ELSE} \\
\text{IF } \text{merging-masking} \\
\text{THEN } \text{DEST}[i+63:i] \text{ remains unchanged} \\
\text{ELSE} \\
\text{zeroing-masking} \\
\text{DEST}[i+63:i] := 0 \\
\text{FI} \\
\text{FI} \\
\text{ENDFOR} \\
\text{DEST[MAXVL-1:VL]} := 0
\]

**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_cvttpd_epi64( __m256d a);
VCVTTPD2QQ __m256i _mm256_mask_cvttpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i _mm256_maskz_cvttpd_epi64( __mmask8 k, __m256d a);
VCVTTPD2QQ __m128i _mm128_cvttpd_epi64( __m128d a);
VCVTTPD2QQ __m128i _mm128_mask_cvttpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i _mm128_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 
\text{IF } \text{EVEX.vvvv} != 1111B.
```

---

VCVTTPD2QQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Quadword Integers
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])
        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/2] := 0

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

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

FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
          Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] :=
          Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI
      FI
    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_cvtpd_epu32(__m512d a);
VCVTTPD2UDQ __m256i _mm512_mask_cvtpd_epu32(__m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_maskz_cvtpd_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_cvtpd_epu32(__m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm256_maskz_cvtpd_epu32(__mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm_mask_cvtpd_epu32(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __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.
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 VCVTTDP2UQQ 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 VCVTTDP2UQQ 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 VCVTTDP2UQQ 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**

**VCVTTDP2UQQ (EVEX encoded versions) when src operand is a register**

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

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

DEST[MAXVL-1:VL] := 0
VCVTTPD2UQQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

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

VCVTTPD2UQQ _mm<size>[_mask[z]]_cvtt[round]pd_epu64
VCVTTPD2UQQ __m512i __mm512_cvtt_roundpd_epu64(__m512d a);
VCVTTPD2UQQ __m512i __mm512_mask_cvtt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_maskz_cvtt_roundpd_epu64(__mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_cvtt_roundpd_epu64((__mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_maskz_cvtt_roundpd_epu64((__mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i __mm256_mask_cvtt_roundpd_epu64(__mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i __mm256_maskz_cvtt_roundpd_epu64(__mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i __mm128_mask_cvtt_roundpd_epu64(__mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i __mm128_maskz_cvtt_roundpd_epu64(__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.
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.128.8F.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.256.8F.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.512.8F.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* ; merging-masking

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

ELSE ; zeroing-masking

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

FI

ENDFOR

\(DEST[\text{MAXVL-1:VL}] := 0\)
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, int sae);
VCVTTPS2UDQ __m256i _mm256_mask_cvttps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i _mm256_maskz_cvttps_epu32( __mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i _mm_mask_cvttps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i _mm_maskz_cvttps_epu32( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions
Invalid, Precision

Other Exceptions
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”; additionally:
#UD If EVEX.vvvv != 1111B.
VCVTPPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F.W0 7A /r VCVTPPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst</td>
<td>A/V/V</td>
<td></td>
<td>AVX512VL AVX512DQ</td>
<td>Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F.W0 7A /r VCVTPPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A/V/V</td>
<td></td>
<td>AVX512VL AVX512DQ</td>
<td>Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F.W0 7A /r VCVTPPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}</td>
<td>A/V/V</td>
<td></td>
<td>AVX512DQ</td>
<td>Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1.</td>
</tr>
</tbody>
</table>

<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**

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**

VCVTPPS2QQ (EVEX encoded versions) when src operand is a register

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

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

\(i := j * 64\)

\(k := j * 32\)

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

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

\(\text{ELSE}\)

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

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

\(\text{ELSE}\)

\(\text{zeroing-masking}\)

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

\(\text{FI}\)

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)
VCVTPPS2QQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1
    i := j * 64
    k := j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                        DEST[i+63:i] := Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
                ELSE
                        DEST[i+63:i] := Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31])
                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
VCVTPPS2QQ __m512i _mm512_cvttps_epi64( __m256 a);
VCVTPPS2QQ __m512i _mm512_mask_cvttps_epi64( __m512i s, __mmask16 k, __m256 a);
VCVTPPS2QQ __m512i _mm512_maskz_cvttps_epi64( __mmask16 k, __m256 a);
VCVTPPS2QQ __m512i _mm512_cvtt_roundps_epi64( __m256 a, int sae);
VCVTPPS2QQ __m512i _mm512_mask_cvtt_roundps_epi64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTPPS2QQ __m256i _mm256_cvttps_epi64( __m256i s, __mmask8 k, __m128 a);
VCVTPPS2QQ __m256i _mm256_mask_cvttps_epi64( __mmask8 k, __m128 a);
VCVTPPS2QQ __m128i _mm_mask_cvtt_roundps_epi64( __mmask8 k, __m128 a);
VCVTPPS2QQ __m128i _mm_maskz_cvtt_roundps_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.
VCVTTPS2UQQ—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</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</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</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 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

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

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

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] := Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VCVTPS2UQQ (EVEX encoded versions) when src operand is a memory source

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

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

VCVTPS2UQQ _mm<size>[z]_cvtt[round]ps_epi64
VCVTPS2UQQ __m512i _mm512_cvttps_epi64(__m256 a);
VCVTPS2UQQ __m512i _mm512_mask_cvttps_epi64(__m512i s, __mmask16 k, __m256 a);
VCVTPS2UQQ __m512i _mm512_maskz_cvttps_epi64(__mmask16 k, __m256 a);
VCVTPS2UQQ __m512i _mm512_cvtt_roundps_epi64(__m256 a, int sae);
VCVTPS2UQQ __m512i _mm512_mask_cvtt_roundps_epi64(__m512i s, __mmask16 k, __m256 a, int sae);
VCVTPS2UQQ __m512i _mm512_maskz_cvtt_roundps_epi64(__mmask16 k, __m256 a, int sae);
VCVTPS2UQQ __m256i _mm256_cvttps_epi64(__m256 i, __mmask8 k, __m128 a);
VCVTPS2UQQ __m256i _mm256_mask_cvttps_epi64(__mmask8 k, __m128 a);
VCVTPS2UQQ __m128i _mm_mask_cvttps_epi64(__m128 i, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i _mm_maskz_cvttps_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.
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.LLIG.F2.0F:W0 78 /r VCVTTSD2USI r32, xmm1/m64{sa}</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>EVEX.LLIG.F2.0F:W1 78 /r VCVTTSD2USI r64, xmm1/m64{sa}</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>
</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_cvttss_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvttss_u64(__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 78 /r VCVTTSS2USI r32, xmm1/m32{sae}</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 78 /r VCVTTSS2USI r64, xmm1/m32{sae}</td>
<td>A</td>
<td>V/N.E.†</td>
<td>AVX512F</td>
<td>Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.</td>
</tr>
</tbody>
</table>

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

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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”.

5-74 Vol. 2C VCVTTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer
VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values

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

The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

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

Operation

VCVTUDQ2PD (EVEX encoded versions) when src operand is a register

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

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

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

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

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

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[i+63:j] remains unchanged* 
ELSE ; zeroing-masking

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

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 _mm128_cvtepu32_pd( __m64i a);
VCVTUDQ2PD __m128d _mm128_mask_cvtepu32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm128_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 VCVTUDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F2.0F.W0 7A /r VCVTUDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A 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 zmm1 with writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F2.0F.W0 7A /r VCVTUDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}</td>
<td>A 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>
<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 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 __m512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __m512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __m512_cvt_roundepu32_ps( __m512i a, int r);
VCVTUDQ2PS __m512 __m512_mask_cvt_roundepu32_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 __m512_maskz_cvt_roundepu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 __m256_cvtepu32_ps( __m256i a);
VCVTUDQ2PS __m256 __m256_mask_cvtepu32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 __m256_maskz_cvtepu32_ps( __mmask8 k, __m256i a);
VCVTUDQ2PS __m128 __m128_cvtepu32_ps( __m128i a);
VCVTUDQ2PS __m128 __m128_mask_cvtepu32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 __m128_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>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 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 (EVAEX encoded version) when src operand is a memory source

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

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] := Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[i+63:i] := Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
        FI
    FI
  FI
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 __m256d _mm256_cvt_roundepu64_ps( __m256i a, int r);
VCVTUQQ2PD __m256d _mm256_mask_cvt_roundepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvt_roundepu64_ps( __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.
VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F2.0F.W1 7A /r VCVTUQQ2PS 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 packed single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {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 single-precision floating-point values in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F2.0F.W1 7A /r VCVTUQQ2PS ymm1 {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 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 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 __m256_mask_cvtepu64_ps(__m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __m256_maskz_cvtepu64_ps(__mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __m256_cvt_roundepu64_ps(__m512i a, int r);
VCVTUQQ2PS __m256 __m256_mask_cvt_roundepu64_ps(__m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 __m256_maskz_cvt_roundepu64_ps(__mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128 __m128_mask_cvtepu64_ps(__m256i a);
VCVTUQQ2PS __m128 __m128_cvt_roundepu64_ps(__m256i a);
VCVTUQQ2PS __m128 __m128_mask_cvt_roundepu64_ps(__m256i a);
VCVTUQQ2PS __m128 __m128_maskz_cvt_roundepu64_ps(__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.
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 VCVTUSI2SD xmm1, xmm2, r/m32</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 VCVTUSI2SD xmm1, xmm2, r/m64(er)</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 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>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</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>VCVTUSI2SS xmm1, xmm2, r/m32[er]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.LLIG.F3.0F.W1 7B /r</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>
<tr>
<td>VCVTUSI2SS xmm1, xmm2, r/m64[er]</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 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 an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

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


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

VCVTUSI2SS _m128 _mm_cvtu32_ss( _m128 s, unsigned a);
VCVTUSI2SS _m128 _mm_cvt_roundu32_ss( _m128 s, unsigned a, int r);
VCVTUSI2SS _m128 _mm_cvtu64_ss( _m128 s, unsigned __int64 a);
VCVTUSI2SS _m128 _mm_cvt_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**

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 42 /r ib VDBPSADBW xmm1 {k1}{z}, xmm2, xmm3/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compute packed SAD word results of unsigned bytes in dword block from xmm2 with unsigned bytes of dword blocks transformed from xmm3/m128 using the shuffle controls in imm8. Results are written to xmm1 under the writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 42 /r ib VDBPSADBW ymm1 {k1}{z}, ymm2, ymm3/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compute packed SAD word results of unsigned bytes in dword block from ymm2 with unsigned bytes of dword blocks transformed from ymm3/m256 using the shuffle controls in imm8. Results are written to ymm1 under the writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 42 /r ib VDBPSADBW zmm1 {k1}{z}, zmm2, zmm3/m512, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compute packed SAD word results of unsigned bytes in dword block from zmm2 with unsigned bytes of dword blocks transformed from zmm3/m512 using the shuffle controls in imm8. Results are written to zmm1 under the writemask k1.</td>
</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>Imm8</td>
</tr>
</tbody>
</table>

**Description**

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

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

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

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

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

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

**VDBPSADBW (EVEX encoded versions)**

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

Selection of quadruplets:

FOR I = 0 to VL step 128

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

END FOR

SAD of quadruplets:

FOR I = 0 to VL step 64

\[
\begin{align*}
&\quad + |\text{SRC1}[I+15:I+8] - \text{TMP1}[I+15:I+8]| \\
&\quad + |\text{SRC1}[I+23:I+16] - \text{TMP1}[I+23:I+16]| \\
&\quad + |\text{SRC1}[I+31:I+24] - \text{TMP1}[I+31:I+24]|
\end{align*}
\]

\[
\begin{align*}
&\quad + |\text{SRC1}[I+23:I+16] - \text{TMP1}[I+31:I+24]| \\
&\quad + |\text{SRC1}[I+15:I+8] - \text{TMP1}[I+23:I+16]| \\
&\quad + |\text{SRC1}[I+31:I+16] - \text{TMP1}[I+31:I+24]|
\end{align*}
\]

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

\[
\begin{align*}
&\quad + |\text{SRC1}[I+47:I+40] - \text{TMP1}[I+39:I+32]| \\
&\quad + |\text{SRC1}[I+55:I+48] - \text{TMP1}[I+47:I+40]| \\
&\quad + |\text{SRC1}[I+63:I+56] - \text{TMP1}[I+55:I+48]|
\end{align*}
\]

ENDFOR

FOR j := 0 TO KL-1

\[
\begin{align*}
&i := j \times 16 \\
&\text{IF k1}[j] \text{ OR *no writemask*} \\
&\quad \text{THEN } \text{DEST}[i+15:i] := \text{TMP_DEST}[i+15:i] \\
&\quad \text{ELSE} \\
&\quad \quad \text{IF *merging-masking*} \\
&\quad \quad \quad \text{THEN *DEST}[i+15:i] remains unchanged* \\
&\quad \quad \quad \text{ELSE *zeroing-masking*} \\
&\quad \quad \quad \quad \text{DEST}[i+15:i] := 0 \\
&\quad \quad \text{FI} \\
&\quad \text{FI} \\
&\text{ENDFOR}
\end{align*}
\]

\[
\text{DEST}[MAXVL-1:VL] := 0
\]
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>Upper part may be overridden if lower has NaN.</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”.
VEXPANDPD—Load Sparse Packed Double-Precision Floating-Point Values from Dense Memory

**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 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__mm128_mask_expand_pd(__m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d__mm128_maskz_expand_pd(__mmask8 k, __m128d a);
VEXPANDPD __m128d__mm128_maskz_expandloadu_pd(__mmask8 k, void * a);
VEXPANDPD __m128d__mm128_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>
<thead>
<tr>
<th>Opcode/Instruction</th>
<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</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}[z], ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}[z], zmm2/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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)**

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

\[k := 0\]

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

\[i := j \times 32\]

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

\[\text{THEN}\]

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

\[k := k + 32\]

\[\text{ELSE}\]

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

\[\text{THEN } *\text{DEST}[i+31:i] \text{ remains unchanged}^*\]

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

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

\[\text{FI}\]

\[\text{FI};\]

\[\text{ENDFOR}\]

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

VEXPANDPS __m512 __m512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 __m512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 __m512_maskz_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 __m512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 __m256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 __m256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 __m256_maskz_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 __m256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 __m128_mask Expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 __m128_mask Expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 __m128_maskz Expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 __m128_maskz Expand_ps( __mmask8 k, __m128 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;
        Fl;
      Fl;
```
Flags Affected
The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

Protected Mode Exceptions
The only exceptions generated for these instructions are those related to illegal addressing of the source operand.
#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
    If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD If the LOCK prefix is used.

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

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

Compatibility Mode Exceptions
Same exceptions as in protected mode.

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

**Opcode/Instruction**

<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[1:0] 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 *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 *merging-masking* ; merging-masking
    THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
    DEST[i+31:i] := 0
  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
        ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI;
ENDFOR

5-102 Vol. 2C
DEST[+63] := 0

FI

DEST[MAXVL-1:256] := 0

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

FOR j := 0 TO 3
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63] := 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.

Intel C/C++ Compiler Intrinsic Equivalent

VEXTRACTF32x4 __m128 _mm512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128 __m512_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 __m512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF128 __m512 _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 != 111B.
VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256,66,0F3A.w0 39 / r ib VEXTRACTI128 xmm1/m128, ymm2, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.</td>
</tr>
<tr>
<td>EVEX.256,66,0F3A.w0 39 / r ib VEXTRACTI32X4 xmm1/m128 {k1}[z], ymm2, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512,66,0F3A.w0 39 / r ib VEXTRACTI32x4 xmm1/m128 {k1}[z], zmm2, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256,66,0F3A.w1 39 / r ib VEXTRACTI64X2 xmm1/m128 {k1}[z], ymm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512,66,0F3A.w1 39 / r ib VEXTRACTI64X2 xmm1/m128 {k1}[z], zmm2, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512,66,0F3A.w0 3B / r ib VEXTRACTI32X8 ymm1/m256 {k1}[z], zmm2, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512,66,0F3A.w1 3B / r ib VEXTRACTI64x4 ymm1/m256 {k1}[z], zmm2, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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.
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
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+63:i] := 0
        FI
      FI;
ENDFOR
DEST[MAXVL-1:256] := 0

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

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

FOR j := 0 TO 3
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
VEXTRACTI128 (memory destination form)
CASE (imm8[0]) OF
  0: DEST[127:0] := SRC1[127:0]
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, __m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_extracti64x2_epi64(__m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_mask_extracti64x2_epi64(__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, __m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_maskz_extracti64x4_epi64(__mmask8 k, __m512i a, const int nidx);

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

### 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.

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<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 54 / m</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 54 / m</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 54 / m</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:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Note:**

- **Op/En**: A
- **Tuple Type**: Full
- **Operand 1**: ModRM:reg (r, w)
- **Operand 2**: EVEX.vvvv (r)
- **Operand 3**: ModRM:r/m (r)
- **Operand 4**: Imm8
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
}

FIXUPIMMEDP (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...)

dest[63:0] := dest[63:0];  ; preserve content of DEST
    0000: dest[63:0] := tsrc[63:0];   ; pass through src1 normal input value, denormal as zero
    0001: dest[63:0] := QNaN(tsrc[63:0]);
    0010: dest[63:0] := qNaN;  
    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:

```
+ INF → #IE
- VE → #IE
- INF → #IE
SNaN → #IE
ONE → #IE
ONE → #ZE
ZERO → #IE
ZERO → #ZE
```

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

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

```
VFIXUPIMMPD __m512d _mm512_fixupimm_pd( __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_pd( __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_fixupimm_round_pd( __m512d a, __m512i tbl, int sae);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_round_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_round_pd( __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m256d _mm256_fixupimm_pd( __m256d a, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_mask_fixupimm_pd(__m256d a, __mmask8 k, __m256d b, __m256i c, int imm8);
VFIXUPIMMPD __m256d _mm256_maskz_fixupimm_pd( __mmask8 k, __m256d a, __m256i c, int imm8);
VFIXUPIMMPD __m128d _mm128_fixupimm_pd( __m128d a, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm128_mask_fixupimm_pd(__m128d a, __mmask8 k, __m128d b, __m128i c, int imm8);
VFIXUPIMMPD __m128d _mm128_maskz_fixupimm_pd( __mmask8 k, __m128d a, __m128d b, 128i c, int imm8);
```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

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

5-114 Vol. 2C

VFIXUPIMMPD—Fix Up Special Packed Float64 Values
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</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 54 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 54 /ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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...)

    ; 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()

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:

![Immediate Control Description Diagram](image)

**Figure 5-10. VFIXUPIMMPS Immediate Control Description**

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

VFIXUPIMMPS_m512_mm512_fixupimm_ps(__m512 a, __m512i tbl, int imm);
VFIXUPIMMPS_m512_mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS_m512_mm512_maskz_fixupimm_ps(__mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS_m512_mm512_fixupimm_round_ps(__m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS_m512_mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS_m512_mm512_maskz_fixupimm_round_ps(__mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS_m256_mm256_fixupimm_ps(__m256 a, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS_m256_mm256_mask_fixupimm_ps(__m256 a, __mmask8 k, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS_m256_mm256_maskz_fixupimm_ps(__mmask8 k, __m256 a, __m256 b, __m256i c, int imm8);
VFIXUPIMMPS_m128_mm128_fixupimm_ps(__m128 a, __m128 b, __m128i c, int imm8);
VFIXUPIMMPS_m128_mm128_mask_fixupimm_ps(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8);
VFIXUPIMMPS_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”. 
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.LLG.66.0F3A.W1 55 /r ib VFIXUPIMMSD xmm1 <a href="z">k1</a>, xmm2, xmm3/m64[saе], imm8</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 \( x=\text{approx}(1/0) \), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

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

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

```plaintext
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
}
```
VFIXUIMMSD—Fix Up Special Scalar Float64 Value

INSTRUCTION SET REFERENCE, V-Z

5-120 Vol. 2C

```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

DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Immediate Control Description:

Immediate Control Description

<table>
<thead>
<tr>
<th>Condition</th>
<th>Control Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ INF</td>
<td>#IE</td>
</tr>
<tr>
<td>- VE</td>
<td>#IE</td>
</tr>
<tr>
<td>- INF</td>
<td>#IE</td>
</tr>
<tr>
<td>SNaN</td>
<td>#IE</td>
</tr>
<tr>
<td>ONE</td>
<td>#IE</td>
</tr>
<tr>
<td>ONE</td>
<td>#ZE</td>
</tr>
<tr>
<td>ZERO</td>
<td>#IE</td>
</tr>
<tr>
<td>ZERO</td>
<td>#ZE</td>
</tr>
</tbody>
</table>

Figure 5-11. VFIXUPIMMSD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMSD __m128d _mm_fixupimm_sd(__m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm.sd(__mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_fixupimm_round_sd(__m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_round_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128i tbl, int imm, int sae);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions”.
VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

**Op/En**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<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>A 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>

**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 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**

```
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_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[6:3]) 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] := MAX_FLOAT;
        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()
```

VFIXUPIMMS—Fix Up Special Scalar Float32 Value
VFIXUPIMMSS (EVEX encoded version)
IF k1[0] OR "no writemask"
    THEN DEST[31:0] := FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
ELSE
    IF "merging-masking"
        THEN "DEST[31:0] remains unchanged"
    ELSE DEST[31:0] := 0 ; zeroing-masking
FI
Ft:
DEST[MAXVL-1:128] := 0

Immediate Control Description:

![Immediate Control Description Diagram](image)

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”.

5-124 Vol. 2C
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<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/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>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.w1 A8 /r VFMADD213PD 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, add to xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.w1 B8 /r VFMADD231PD 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, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.w1 98 /r VFMADD132PD 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, add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.w1 A8 /r VFMADD213PD 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, add to ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.w1 B8 /r VFMADD231PD 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, add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w1 98 /r VFMADD132PD 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, add to xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w1 A8 /r VFMADD213PD 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, add to xmm3/m128/m64bcst and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w1 B8 /r VFMADD231PD 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, add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 98 /r VFMADD132PD 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, add to ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 A8 /r VFMADD213PD 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, add to ymm3/m256/m64bcst and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 B8 /r VFMADD231PD 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, add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w1 98 /r VFMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B/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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w1 A8 /r VFMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B/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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}</td>
<td>B/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>
<td></td>
</tr>
</tbody>
</table>
VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

**Instruction Set Reference, V-Z**

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

**EVEX encoded versions:** The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

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

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

**VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)**

If (VEX.128) then
    MAXNUM := 2
ElseIf (VEX.256)
    MAXNUM := 4
EndIf
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
EndIf

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

If (VEX.128) then
    MAXNUM := 2
ElseIf (VEX.256)
    MAXNUM := 4
EndIf
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
EndIf

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

If (VEX.128) then
    MAXNUM := 2
ElseIf (VEX.256)
    MAXNUM := 4
EndIf
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
EndIf
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;
    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[i+63:i])
                ELSE
                    DEST[i+63:i] :=
                        RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
                FI
            FI;
    FI;
ENDFOR

DEST[MAXVL-1:VL] := 0
VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)  
(KL, VL) = (2, 128), (4, 256), (8, 512)  
IF (VL = 512) AND (EVEX.b = 1)  
THEN  
SET_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  

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)  
(KL, VL) = (2, 128), (4, 256), (8, 512)  
FOR j := 0 TO KL-1  
i := j * 64  
IF k1[j] OR *no writemask*  
THEN  
   IF (EVEX.b = 1)  
      THEN  
         DEST[i+63:i] :=  
         RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[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  
ENDFOR  
DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __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_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d _mm256d_fmadd_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 xmm2, add to 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>VEX.256.66.0F38.0 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>AVX512VL 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>AVX512VL 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>AVX512VL 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</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, 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</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 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</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 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 destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMADD132PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD213PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFMADD231PS:** Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

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

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

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

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[31:0])
                ELSE
                    DEST[i+31:i] :=
                        RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN
                    *DEST[i+31:i] remains unchanged*
                ELSE
                    ; zeroing-masking
                    DEST[i+31:i] := 0
            FI
        FI
ENDFOR
DEST[MAXVL-1:VL] := 0
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;
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_fadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 __mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 __mm256_fadd_ps(__m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_fmadd_ps(__m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 __mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 __mm128_fadd_ps(__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm128_fmadd_ps(__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm128_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm128_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm128_mask3_fmadd_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”.
**VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double-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-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]*DEST[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_fmadd_round_sd(__m128d a, __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_fmadd_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”.
VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add 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-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.
EVAEX 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])

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

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])

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

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])

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, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_maskz_fmadd_ss(__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, __m128 b, __m128 c, __mmask8 k, 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, int r);

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”. 
<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<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 VFMADDSUB132PD 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/subtract elements in ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.w1 A6 /r VFMADDSUB213PD 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/subtract elements in ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.w1 B6 /r VFMADDSUB231PD 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/subtract elements in ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w1 96 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w1 A6 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w1 B6 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 96 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 A6 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 B6 /r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Instruction Set Reference, V-Z**

### 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.512.66.0F38.W1 A6 /r VFMADDSUB213PD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst[er]</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 B6 /r VFMADDSUB231PD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst[er]</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 96 /r VFMADDSUB132PD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst[er]</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

**VFMADDSUB132PD**: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point elements 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
        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
    FI
ENDFOR

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

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

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

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

VFMADDSUBxDDD __m512d __mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxDDD __m512d __mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxDDD __m512d __mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxDDD __m512d __mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxDDD __m512d __mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxDDD __m512d __mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxDDD __m512d __mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxDDD __m256d __mm256_fmaddsub_pd(__m256d a, __m256d b, __m256d c);
VFMADDSUBxDDD __m256d __mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxDDD __m256d __mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxDDD __m256d __mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxDDD __m128d __mm128_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxDDD __m128d __mm128_mask_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxDDD __m128d __mm128_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxDDD __m128d __mm128_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxDDD __m128d __mm128_mask_fmaddsub_round_pd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDSUBxDDD __m128d __mm128_maskz_fmaddsub_round_pd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDSUBxDDD __m128d __mm128_mask3_fmaddsub_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”.
### 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.128.66.0F38.0A B6 /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/mem, add/subtract elements in xmm3 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.0B 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.256.66.0F38.0A B6 /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/mem, add/subtract elements in ymm3 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.0A A6 /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 xmm1 and xmm2/mem, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.0B B6 /r VFMADDSUB231PS xmm1, xmm2, xmm3/m256</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>EVEX.256.66.0F38.0A A6 /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 ymm1 and ymm2/mem, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.0B 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.512.66.0F38.0A A6 /r VFMADDSUB213PS 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/mem, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.0B 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/mem, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.0C C6 /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/mem, add/subtract 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>EVEX.vvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

VFMADD132PS: 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 second 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).

VFMADD213PS: 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).

VFMADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

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

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

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

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

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

**VFMADDSUB132PS DEST, SRC2, SRC3**

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

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

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

**VFMADDSUB213PS DEST, SRC2, SRC3**

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

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

IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI
VFMADDSUB231PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM -1{
    n := 64*i;
    DEST[n+31:n] := RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
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
                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*
                THEN
                    DEST[i+31:i] remains unchanged*
            ELSE
                    DEST[i+31:i] := 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

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

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
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

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

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

VFMADDUBxXPS __m512 __mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDUBxXPS __m512 __mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDUBxXPS __m512 __mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDUBxXPS __m512 __mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDUBxXPS __m512 __mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDUBxXPS __m512 __mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDUBxXPS __m512 __mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDUBxXPS __m512 __mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDUBxXPS __m256 __mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDUBxXPS __m256 __mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDUBxXPS __m256 __mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDUBxXPS __m128 __mm128_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDUBxXPS __m128 __mm128_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDUBxXPS __m128 __mm128_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDUBxXPS __m256 __mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see 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 /rVFMSUBADD213PD 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 /rVFMSUBADD231PD 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 /rVFMSUBADD132PD ymm1, xmm2, ymm3/m256</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 ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 A7 /rVFMSUBADD213PD ymm1, xmm2, ymm3/m256</td>
<td>A V/V</td>
<td>FMA</td>
<td>Multiply packed double-precision floating-point values from ymm1 and xmm2, subtract/add elements in ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 B7 /rVFMSUBADD231PD ymm1, xmm2, 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 /rVFMSUBADD132PD 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 /rVFMSUBADD213PD 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 /rVFMSUBADD231PD 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 /rVFMSUBADD132PD 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 /rVFMSUBADD213PD 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>
<tr>
<td>EVEX.256.66.0F38.W1 B7 /rVFMSUBADD231PD 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/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double-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.512.66.0F38.W1 97 /r VFMSUBADD132PD 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, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A7 /r VFMSUBADD213PD 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, subtract/add elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 B7 /r VFMSUBADD231PD 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, 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

**VFMSUBADD132PD**: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD213PD**: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

**VFMSUBADD231PD**: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

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

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

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

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

**VFMSUBADD132PD DEST, SRC2, SRC3**

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

**VFMSUBADD213PD DEST, SRC2, SRC3**

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

**VFMSUBADD231PD DEST, SRC2, SRC3**

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

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
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE
      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
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;  
  ELSE  
  IF *merging-masking*  
  THEN *DEST[i+63:i] remains unchanged*  
  ELSE  
      DEST[i+63:i] := 0  
  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, __mmask8 k);
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 __m256d __mm256_fmsubadd_pd(__m256d a, __m256d b, __m256d c);

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

Other Exceptions
VEX-encoded instructions, see 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.128.66.0F38.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.128.66.0F38.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.128.66.0F38.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.256.66.0F38.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.256.66.0F38.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.256.66.0F38.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.128.66.0F38.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.128.66.0F38.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.128.66.0F38.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.256.66.0F38.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.256.66.0F38.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.256.66.0F38.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.512.66.0F38.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{er}, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.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{er} and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.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>EVEX.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).

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

**VFMSUBADD132PS DEST, SRC2, SRC3**

If (VEX.128) THEN
    MAXNUM := 2
Elseif (VEX.256)
    MAXNUM := 4
Fi

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

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* ; 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
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 _mm_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBADDxxxPS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see 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
### 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>VEX.128.66.0F38.W1 9A /r VFMSUB132PD 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, subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 AA /r VFMSUB213PD 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, subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 BA /r VFMSUB231PD 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, subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 9A /r VFMSUB132PD 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, subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 AA /r VFMSUB213PD 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, subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 BA /r VFMSUB231PD 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, subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 9A / r VFMSUB132PD 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/mem, subtract xmm2 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 AA / r VFMSUB213PD 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, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 BA / r VFMSUB231PD 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/mem, subtract xmm1 and put result in xmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 9A / r VFMSUB132PD 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/mem, subtract ymm2 and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 AA / r VFMSUB213PD 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, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 BA / r VFMSUB231PD 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/mem, subtract ymm1 and put result in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 9A / r VFMSUB132PD 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/mem, subtract zmm2 and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 AA / r VFMSUB213PD 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, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 BA / r VFMSUB231PD 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/mem, subtract zmm1 and put result in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>
VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

**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 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 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
ENDFOR
DEST[MAXVL-1:VL] := 0

VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

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

(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_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

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

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
    THEN
        IF (EVEX.b = 1)
            THEN
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0]) +31:i)
            ELSE
                DEST[i+63:i] :=
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI;
    ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)

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

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

THEN

    SET_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, __m512d b, __m512d c, __mmask8 k);
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, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_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_pd(__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);

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

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/E</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.w0 9A /r VFMSUB132PS 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 xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.w0 AA /r VFMSUB213PS 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 xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.w0 BA /r VFMSUB231PS 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 xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.w0 9A /r VFMSUB132PS 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 ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.w0 AA /r VFMSUB213PS 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 ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.00 BA /r VFMSUB231PS 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 ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.w0 9A /r VFMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</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, subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.w0 AA /r VFMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.w0 BA /r VFMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.</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 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
End

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
End

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

If VEX.128 then
   MAXNUM := 2
Elseif VEX.256
   MAXNUM := 4
End

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
End

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

If VEX.128 then
   MAXNUM := 2
Elseif VEX.256
   MAXNUM := 4
End

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
End
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[31:0] - SRC2[i+31:i])
      ELSE
        DEST[i+31:i] :=
          RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
THEN
SET_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
ENDIF;
ENDFOR
DEST[MAXVL-1:VL] := 0

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

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
i := j * 32
IF k1[j] OR *no writemask*
THEN
IF (EVEX.b = 1)
THEN
DEST[i+31:i] :=
RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
ELSE
DEST[i+31:i] :=
RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+31:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+31:i] := 0
FI
ENDIF;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)

IF (VL = 512) AND (EVEX.b = 1)
THEN
    SET_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:j] :=
        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:j] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:j] := 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:j] :=
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
            ELSE
                DEST[i+31:j] :=
                    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:j] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:j] := 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 __m128 __mm128_fmsub_ps(__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m256 __mm256_fmsub_ps(__m256 a, __m256 b, __m256 c);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.
VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-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>Tuple 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 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 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

\[
\text{SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION}(\text{EVEX.RC})
\]

ELSE

\[
\text{SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION}(\text{MXCSR.RC})
\]

FI;

IF k1[0] or *no writemask*

THEN \[\text{DEST}[63:0] := \text{RoundFPControl}\left(\text{DEST}[63:0] \times \text{SRC3}[63:0] - \text{SRC2}[63:0]\right)\]

ELSE

IF *merging-masking* ; merging-masking

THEN \[\text{DEST}[63:0] \text{ remains unchanged}\]

ELSE ; zeroing-masking

THEN \[\text{DEST}[63:0] := 0\]

FI;

FI;

\[
\text{DEST}[127:64] := \text{DEST}[127:64]
\]

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

**VFMSUB213SD 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 \[\text{DEST}[63:0] := \text{RoundFPControl}\left(\text{SRC2}[63:0] \times \text{DEST}[63:0] - \text{SRC3}[63:0]\right)\]

ELSE

IF *merging-masking* ; merging-masking

THEN \[\text{DEST}[63:0] \text{ remains unchanged}\]

ELSE ; zeroing-masking

THEN \[\text{DEST}[63:0] := 0\]

FI;

FI;

\[
\text{DEST}[127:64] := \text{DEST}[127:64]
\]

\[
\text{DEST}[\text{MAXVL}-1:128] := 0
\]
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(Destination[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] := DEST[127:64]
DEST[MAXVL-1:128] := 0

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

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

Intel C/C++ Compiler Intrinsic Equivalent
VFMSUBxxxSD __m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d _mm_mask_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

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

Other Exceptions
VEX-encoded instructions, see 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 with the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.
EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

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

**VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)**

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

THEN

SET_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

**VFMSUB213SS 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
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, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_ss(__m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __m128 k);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __m128 k, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_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 /r VFNMADD132PD 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 /r VFNMADD213PD 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 /r VFNMADD231PD 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 /r VFNMADD132PD 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 /r VFNMADD213PD 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 /r VFNMADD231PD 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 /r VFNMADD132PD 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 /r VFNMADD213PD 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 /r VFNMADD231PD 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>
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).

<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>
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[63:0]) + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] :=
                    RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
                    FI;
            ELSE
                IF *merging-masking* ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                ELSE ; zeroing-masking
                    DEST[i+63:i] := 0
                FI
            FI;
        ENDFOR

DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent
VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd(__m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd(__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd(__m256d a, __m256d b, __m256d c);

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

Other Exceptions
VEX-encoded instructions, see Table 2-19, “Type 2 Class Exception Conditions”.
EVEX-encoded instructions, see Table 2-46, “Type E2 Class Exception Conditions”.
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.w0 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[1: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[1: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[1: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[1: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[1: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[1: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[1: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[1: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[1: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 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) 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.128 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 ZMM/YMM/XMM 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 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 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])
        ENDIF;
      ELSE
        IF *merging-mask* ; merging-mask
          THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-mask
          DEST[i+31:i] := 0
        ENDIF;
    ENDIF;
  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-mask* ; merging-mask
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-mask
        DEST[i+31:i] := 0
      ENDIF;
    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_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
VFNMADDxxxPS __m512_mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512_mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512_mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512_mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512_mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512_mask_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m256_fnmadd_ps (__m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256_mask_fnmadd_ps (__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256_mask3_fnmadd_ps (__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMADDxxxPS __m128_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128_mask_fnmadd_ps (__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxPS __m128_mask3_fnmadd_ps (__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxPS __m128_maskz_fnmadd_ps (__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m256_fnmadd_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”.
VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W1 9D /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 AD /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 BD /r</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 9D /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 AD /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 BD /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and 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) VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Tuple 1 Scalar</td>
<td>ModRM:reg (r, w) EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td></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”.

VFNMADD132SD/VFNADD213SD/VFNADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values  Vol. 2C  5-219
VFNMADD132SS/VFMADD213SS/VFMADD231SS—Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<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>VEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the 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 (EVEX 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 (EVEX 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 (EVEX 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
VFNMADDxxxSS __m128 _mm_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxSS __m128 _mm_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxSS __m128 _mm_fnmadd_round_ss (__m128 a, __m128 b, __m128 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”.

INSTRUCTION SET REFERENCE, V-Z
## 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 VFNMSUB132PD 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 subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 AE /r VFNMSUB213PD 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 subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 BE /r VFNMSUB231PD 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 subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 9E /r VFNMSUB132PD 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 subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 AE /r VFNMSUB213PD 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 subtract ymm3/mem and put result in ymm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 BE /r VFNMSUB231PD 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 subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 9E /r VFNMSUB132PD 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, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 AE /r VFNMSUB213PD 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, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 BE /r VFNMSUB231PD 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, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 9E /r VFNMSUB132PD 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, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 AE /r VFNMSUB213PD 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, negate the multiplication result and subtract ymm3/m256/m64bcst and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 BE /r VFNMSUB231PD 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, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 9E /r VFNMSUB132PD 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 subtract zmm2 and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 AE /r VFNMSUB213PD 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 subtract zmm3/m512/m64bcst and put result in zmm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 BE /r VFNMSUB231PD 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 subtract zmm1 and put result in zmm1.</td>
</tr>
</tbody>
</table>
Instruction Set Reference, V-Z

**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**

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 256-bit memory location 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 128-bit memory location 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

5-224 Vol. 2C — VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point
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[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;
  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+63:0])
                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

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

VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point
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 _mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_round_pd(__mmask8 k, __mmask8 k, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m256d _mm256_fnmsub_pd(__m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d _mm256_fnmsub_maskz_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m256d _mm256_fnmsub_maskz_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k, __m256d c, __m256d c);
VFNMSUBxxxPD __m128d _mm128_fnmsub_pd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm128_fnmsub_maskz_fnmsub_pd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm128_fnmsub_maskz_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxPD __m128d _mm128_fnmsub_maskz_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k, __m128d c, __m128d 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".
### 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 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>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 AE /r VFNMSUB213PS 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, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 BE /r VFNMSUB231PS 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, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 9E /r VFNMSUB132PS 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, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 AE /r VFNMSUB213PS 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, negate the multiplication result and subtract ymm3/mem and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 BE /r VFNMSUB231PS 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, negate the multiplication result and subtract ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 9E /r VFNMSUB132PS 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, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 AE /r VFNMSUB213PS 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, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 BE /r VFNMSUB231PS 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, negate the multiplication result subtract add to xmm1 and put result in xmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 9E /r VFNMSUB132PS 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, negate the multiplication result and subtract ymm2 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 AE /r VFNMSUB213PS 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, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 BE /r VFNMSUB231PS 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, negate the multiplication result subtract add to ymm1 and put result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 9E /r VFNMSUB132PS 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, negate the multiplication result and subtract zmm2 and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 AE /r VFNMSUB213PS 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, negate the multiplication result and subtract zmm3/m512/m32bcst and put result in zmm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 BE /r VFNMSUB231PS 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, negate the multiplication result subtract add to zmm1 and put result in zmm1.</td>
<td></td>
</tr>
</tbody>
</table>
Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

**VFNMSUB132PS**:
Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB213PS**:
Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

**VFNMSUB231PS**:
Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

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

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

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

**Operation**

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

**VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)**

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

For i = 0 to MAXNUM-1 {
  n := 32*i;
  DEST[n+31:n] := RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}

IF (VEX.128) THEN
  DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
  DEST[MAXVL-1:256] := 0
FI
VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
   MAXNUM := 2
ELSEIF (VEX.256)
   MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
   n := 32*i;
   DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI

VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)
IF (VEX.128) THEN
   MAXNUM := 2
ELSEIF (VEX.256)
   MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
   n := 32*i;
   DEST[n+31:n] := RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
IF (VEX.128) THEN
   DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
   DEST[MAXVL-1:256] := 0
FI

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
   THEN
      SET_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;
    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;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

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

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
INSTRUCTION SET REFERENCE, V-Z

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 _mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m256 _mm256_mask_fnmsub_ps(__m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_maskz_fnmsub_ps(__m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 _mm_mask_fnmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm_maskz_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m128 _mm_maskz_fnmsub_round_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxPS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

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

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

5-234   Vol. 2C   VFNMSUB232PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Val-
### VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.</td>
</tr>
<tr>
<td>VEX.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1, xmm2, xmm3/m64</td>
<td>A</td>
<td>V/V</td>
<td>FMA</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1 (k1)[z], xmm2, xmm3/m64[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1 (k1)[z], xmm2, xmm3/m64[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.</td>
</tr>
<tr>
<td>EVEX.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1 (k1)[z], xmm2, xmm3/m64[er]</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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;
    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;
    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, __m128d b, __m128d c);  
VFNMSUBxxSD __m128d _mm_maskz_fnmsub_sd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxSD __m128d _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c);
VFNMSUBxxSD __m128d _mm_mask_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxSD __m128d _mm_maskz_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, 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”.
Instruction Set Reference, V-Z

VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

**Opcode/Instruction** | **Op/En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description** |
--- | --- | --- | --- | --- |
VEX.LIG.66.0F38.W0 9F /r | A | V/V | FMA | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1. |
VEX.LIG.66.0F38.W0 AF /r | A | V/V | FMA | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1. |
VEX.LIG.66.0F38.W0 BF /r | A | V/V | FMA | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1. |
EVEX.LIG.66.0F38.W0 9F /r | B | V/V | AVX512F | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1. |
EVEX.LIG.66.0F38.W0 AF /r | B | V/V | AVX512F | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1. |
EVEX.LIG.66.0F38.W0 BF /r | B | V/V | AVX512F | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1. |

**Instruction Operand Encoding**

<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>ModRMreg/r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Tuple1 Scalar</td>
<td>ModRMreg (r, w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg/r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

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

**VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)**

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

**VFNMSUB213SS 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
**INSTRUCTION SET REFERENCE, V-Z**

**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**
VFNMSUBxxxxSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxxSS __m128 _mm_mask_fnmsub_ss(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxxSS __m128 _mm_maskz_fnmsub_ss(__m128 a, __m128 b, __m128 c);
VFNMSUBxxxxSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxxSS __m128 _mm_mask_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxxSS __m128 _mm_maskz_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxxSS __m128 _mm_maskz_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);

**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”.
VFPCLASSPD—Tests Types Of a Packed Float64 Values

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.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 FPCLASSPD instruction checks the packed double precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-4.

![Figure 5-13. Imm8 Byte Specifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS](image)

Table 5-4. Classifier Operations for VFPCLASSPD/SD/PS/SS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>QNaN</td>
<td>PosZero</td>
<td>NegZero</td>
<td>PosINF</td>
<td>NegINF</td>
<td>Denormal</td>
<td>Negative</td>
<td>SNAN</td>
</tr>
<tr>
<td>Classifier</td>
<td>Checks for QNaN</td>
<td>Checks for +0</td>
<td>Checks for -0</td>
<td>Checks for +INF</td>
<td>Checks for -INF</td>
<td>Checks for Denormal</td>
<td>Checks for Negative finite</td>
<td>Checks for SNaN</td>
</tr>
</tbody>
</table>

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

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

CheckFPClassDP (tsrc[63:0], imm8[7:0]) {

    //* Start checking the source operand for special type */
    NegNum := tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber := ExpAllZeros AND MantAllZeros
    SignalingBit := tsrc[51];

    sNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllZeros AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllZeros) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND 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

VFPCLASSPD __mmask8 __mm512_fpclass_pd_mask(__m512d a, int c)
VFPCLASSPD __mmask8 __mm512_mask_fpclass_pd_mask(__mmask8 m, __m512d a, int c)
VFPCLASSPD __mmask8 __mm256_fpclass_pd_mask(__m256d a, int c)
VFPCLASSPD __mmask8 __mm256_mask_fpclass_pd_mask(__mmask8 m, __m256d a, int c)
VFPCLASSPD __mmask8 __mm128_fpclass_pd_mask(__m128d a, int c)
VFPCLASSPD __mmask8 __mm_mask_fpclass_pd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None

Other Exceptions

See 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</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</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</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 VFPCLASSPS 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

CheckFPClassSP (tsrc[31:0], imm8[7:0])

/* Start checking the source operand for special type */
NegNum := tsrc[31];
IF (tsrc[30:23]=0Fh) 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.
VFPCLASSSD—Tests Types Of a Scalar Float64 Values

### Description
The VFPCLASSSD 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.

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

### Operation
CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    NegNum := tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes := 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros := 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros := 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros := 1;
    FI;
    ZeroNumber := ExpAllZeros AND MantAllZeros
    SignalingBit := tsrc[51];

    sNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res := ExpAllOnes AND NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res := NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res := NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res := NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res := NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res := ExpAllZeros AND NOT(MantAllZeros); // denorm
    FinNeg_res := NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = (imm8[0] AND qNaN_res) OR (imm8[1] AND Pzero_res) OR
    Return bResult;
} //* end of CheckFPClassDP() */
VFCLASSSD (EVEX encoded version)
IF k1[0] OR *no writemask*
    THEN DEST[0] :=
        CheckFPClassDP(SRC1[63:0], imm8[7:0])
    ELSE DEST[0] := 0 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VFCLASSSD __mmask8 __mm_fclass_sd_mask(__m128d a, int c)
VFCLASSSD __mmask8 __mm_mask_fclass_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 != 111B.
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 VFPCLASSSS instruction checks the low single-precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1:1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-4.

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; 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;

5-248  Vol. 2C  VFPCLASSSS—Tests Types Of a Scalar Float32 Values
VFCLASSSS (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
VFCLASSSS __mmask8 __m128 __mmask8 __mm_class_ss_mask(__m128 a, int c)
VFCLASSSS __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.
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.
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/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, __m128i 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’d 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(VINDEX[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)**

```c
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)**

```c
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

**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.

---

**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>
• 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 * 32
  IF k1[j]
    THEN DEST[i+31:i] :=
        MEM[BASE_ADDR +
        SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
    k1[j] := 0
    ELSE *DEST[i+31:i] := remains unchanged*
    FI;
  ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL] := 0

**VGATHERDPD (EVEX encoded version)**

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

FOR j := 0 TO KL-1
  i := j * 64
  k := j * 32
  IF k1[j]
    THEN DEST[i+63:i] := MEM[BASE_ADDR +
        SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
    k1[j] := 0
    ELSE *DEST[i+63:i] := remains unchanged*
    FI;
  ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[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_mm512_mask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m512d__mm512_mask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512__mm512_i32gather_ps(__m512i vdx, void * base, int scale);
VGATHERDPS __m512__mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256__mm256__mm512_mask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128__mm128__mm_mask_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

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64y</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 93 /vsib VGATHERQPD xmm1 {k1}, vm64x</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.</td>
</tr>
</tbody>
</table>

### 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 \(k1[j] \text{ OR } \text{no writemask}\)  
    THEN \(\text{DEST}[i+31:i] := \text{MEM}[\text{BASE_ADDR} + (\text{VINDEX}[k+63:k]) \times \text{SCALE} + \text{DISP}]\)  
    \[k1[j] := 0\]  
    ELSE \(\text{DEST}[i+31:i] := \text{remains unchanged}\)  
  FI;  
ENDFOR  
\(k1[\text{MAX_KL-1:KL}] := 0\)  
\(\text{DEST}[\text{MAXVL-1:VL/2}] := 0\)

**VGATHERQPD (EVEX encoded version)**

(\(KL, VL\)) = (2, 128), (4, 256), (8, 512)  
FOR \(j := 0\) TO \(KL-1\)  
  \[i := j \times 64\]  
  IF \(k1[j] \text{ OR } \text{no writemask}\)  
    THEN \(\text{DEST}[i+63:i] := \text{MEM}[\text{BASE_ADDR} + (\text{VINDEX}[i+63:i]) \times \text{SCALE} + \text{DISP}]\)  
    \[k1[j] := 0\]  
    ELSE \(\text{DEST}[i+63:i] := \text{remains unchanged}\)  
  FI;  
ENDFOR  
\(k1[\text{MAX_KL-1:KL}] := 0\)  
\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)
**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 __mm128_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_i64gather_ps(__m512i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_mask_i64gather_ps(__m256 d, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 __mm256_mask_i64gather_ps(__m128 d, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 __mm128_mask_i64gather_ps(__m128 d, __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 AVX512F</td>
<td>Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 42 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 42 /r</td>
<td>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>
</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>
<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])**

```plaintext
{  // 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])**

```plaintext
{  
    Src.sign := 0;  // Zero out sign bit
    Src.exp := SRC[62:52];
    Src.fraction := SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {  
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (Src = +INF) RETURN (Src);
    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    ELSE  // check if denormal operand (notice that MXCSR.DAZ = 0)
    {  
        IF ((Src.exp = 0) AND (Src.fraction != 0))
        {  
            TMP[63:0] := NormalizeExpTinyDPFP(SRC[63:0]) ;  // Get Normalized Exponent
            Set #DE
        }
        ELSE  // exponent value is correct
        {  
            TMP[63:0] := (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction);
        }
        TMP := SAR(TMP, 52);  // Shift Arithmetic Right
        TMP := TMP – 1023;  // Subtract Bias
        Return CvtI2D(TMP);  // Convert INT to Double-Precision FP number
    }
}
```
VGETEXPPD (EVEX encoded versions)

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd( __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd( __mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd( __mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd( __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions"; additionally:

#UD If EVEX.vvvv != 111B.
VGETEXPSS—Convert Exponents of Packed SP FP Values to SP FP Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.{128,256,512}.66.0F38.W0 42 /r VGETEXPPS xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.{128,256,512}.66.0F38.W0 42 /r VGETEXPPS ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
<tr>
<td>EVEX.{128,256,512}.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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>

**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.

**Table 5-6. VGETEXPSS/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.

```
Figure 5-14. VGETEXPPS Functionality On Normal Input values

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>exp</td>
<td>Fraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>----------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Src = 2^1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR Src, 23 = 080h</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bias</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmp - Bias = 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cvt_PI2PS(01h) = 2^0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 5-14. VGETEXPPS Functionality On Normal Input values

**Operation**

```c
NormalizeExpTinySPFP(SRC[31:0])
{
    // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit := 0;
    Dst.exp := 1;
    Dst.fraction := SRC[22:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit := Dst.fraction[22]; // Get the fraction MSB
        Dst.fraction := Dst.fraction << 1; // One bit shift left
        Dst.exp--; // Decrement the exponent
    }
    Dst.fraction := 0; // zero out fraction bits
    Dst.sign := 1; // Return negative sign
    TMP[31:0] := MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
    Return (TMP[31:0]);
}
```

ConvertExpSPFP(SRC[31:0])
{
    Src.sign := 0; // Zero out sign bit
    Src.exp := SRC[30:23];
    Src.fraction := SRC[22:0]; // Check for NaN
    IF (SRC = NaN)
    {
        IF (SRC = SNAN) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (Src = +INF) RETURN (Src);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    }
    ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
    {
```
IF ((Src.exp = 0) AND (Src.fraction != 0))
{
    TMP[31:0] := NormalizeExpTinySPFP(SRC[31:0]);  // Get Normalized Exponent
    Set #DE
}
ELSE  // exponent value is correct
{
    TMP[31:0] := (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction);
}
TMP := SAR(TMP, 23);  // Shift Arithmetic Right
TMP := TMP – 127;  // Subtract Bias
Return 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-mask*  // 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>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>A</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] := ConvertExpDPFP(SRC2[63:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
        DEST[63:0] := 0
    FI
FI
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0
Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSD __m128d _mm_getexp_sd( __m128d a, __m128d b);
VGETEXPSD __m128d _mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_maskz_getexp_sd( __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_getexp_round_sd( __m128d a, __m128d b);
VGETEXPSD __m128d _mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_maskz_getexp_round_sd( __mmask8 k, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See 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.LLIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}[z], xmm2, xmm3/m32{sae}</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.vvvv (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 \( \text{floor}(x) \) stands for maximal integer not exceeding real number \( x \).

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

### Operation

// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS  
// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

**VGETEXPSS (EVEX encoded version)**

IF k1[0] OR *no writemask*
    THEN DEST[31:0] :=
        ConvertExpDPFP(SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] := 0
        FI
    FI;
ENDFOR
DEST[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

### 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.

#### Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input DP FP value \( x \), The conversion operation is:

\[
\text{GetMant}(x) = \pm 2^k |x.\text{significand}|
\]

where:

\[1 \leq |x.\text{significand}| < 2\]

Unbiased exponent \( k \) 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 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. 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>+∞</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>
</tbody>
</table>
| -∞      | IF (SC[1]) THEN QNaN_Indefinite ELSE { 
|         | IF (SC[0]) THEN +1.0 ELSE -1.0   | Ignore \( \text{interv} \)              |
| negative| SC[1] ? QNaN_Indefinite : Getmant(SRC) \(^1\) | If (SC[1]) then \#IE                  |

**NOTES:**
1. In case \( SC[1] == 0 \), the sign of \( \text{Getmant}(\text{SRC}) \) 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(__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(__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

**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.128.66.0F3A.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]
    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 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
return dst

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 _mm_getmant_ps( __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_mask_getmant_ps( __m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_maskz_getmant_ps( __mmask8 k, __m128 a, enum intv, enum sgn);

SIMD Floating-Point Exceptions
Denormal, Invalid

Other Exceptions
See Table 2-46, "Type E2 Class Exception Conditions"; additionally:
#UD If EVEX.vvvv != 1111B.
VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

**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 27 /r ib VGETMANTSD xmm1 [k1]{z}, xmm2, xmm3/m64{sae}, imm8</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:

\[
GetMant(x) = \pm 2^k |x.\text{significand}|
\]

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*
            THEN *DEST[63:0] remains unchanged*
        ELSE
            DEST[63:0] := 0
        FI
FI
Dest[127:64] := SRC1[127:64]
Dest[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VGETMANTSD __m128d _mm_getmant_sd( __m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd( __m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);

SIMD Floating-Point Exceptions
Denormal, Invalid

Other Exceptions
See Table 2-47, “Type E3 Class Exception Conditions”.

VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar
VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.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:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to SP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

\[
GetMant(x) = \pm 2^k |x.\text{significand}|
\]

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</td>
<td>V/V</td>
<td>AVX</td>
<td>Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 18 / r ib VINSERTF32X4 ymm1 [k1][z], ymm2, xmm3/m128, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 18 / r ib VINSERTF32X4 zmm1 [k1][z], zmm2, xmm3/m128, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 18 / r ib VINSERTF64X2 ymm1 [k1][z], ymm2, xmm3/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 18 / r ib VINSERTF64X2 zmm1 [k1][z], zmm2, xmm3/m128, imm8</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 1A / r ib VINSERTF32X8 zmm1 [k1][z], zmm2, ymm3/m256, imm8</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 1A / r ib VINSERTF64X4 zmm1 [k1][z], zmm2, ymm3/m256, imm8</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 256 bits of packed double-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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] := TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VINSERTF64x2 (EVEX encoded versions)**

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

TEMP_DEST[VL-1:0] := SRC1[VL-1:0]

IF VL = 256
    CASE (imm8[0]) OF
        0: 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] := 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
IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
    DEST[i+63:i] := 0
FI
FI;
ENDIFOR
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: TEMP_DEST[255:0] := SRC2[255:0]
  1: TEMP_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] := 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

**VINSERTF64x4 (EVEX.512 encoded version)**

VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
  0: TEMP_DEST[255:0] := SRC2[255:0]
  1: TEMP_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] := 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
**VINSERTF128 (VEX encoded version)**

\[
\text{DEST} := \text{TEMP}
\]

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

VINSERTF32x4 __m512 _mm512_insertf32x4( __m512 a, __m128 b, int imm);
VINSERTF32x8 __m512 _mm512_insertf32x8( __m512 a, __m256 b, int imm);
VINSERTF32x4 __m512 _mm512_mask_insertf32x4(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x8 __m512 _mm512_mask_insertf32x8(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF64x2 __m512d _mm512_insertf64x2( __m512d a, __m128d b, int imm);
VINSERTF64x4 __m512d _mm512_insertf64x4( __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512_mask_insertf64x2(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x4 __m512d _mm512_mask_insertf64x4(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF128d __m256d _mm256_insertf128_pd( __m256d a, __m128d b, int offset);
VINSERTF128d __m256d _mm256_insertf128_ps( __m256d a, __m128d b, int offset);

**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".

---

**VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values**
VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.66.0F3A.W0 38 / r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 38 / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 38 / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 38 / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 38 / r ib</td>
<td>B</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 3A / r ib</td>
<td>D</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 3A / r ib</td>
<td>C</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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>EVEX.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>EVEX.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>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.
**Operation**

**VINSERTI32x4 (EVEX encoded versions)**

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

TEMP_DEST[VL-1:0] := SRC1[VL-1:0]

IF VL = 256
  CASE (imm8[0]) OF
    0: 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

DEST[MAXVL-1:VL] := 0
IF *merging-masking* ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
    DEST[i+63:i] := 0
FI

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

VINSERTI32x8 (EVEX.U1.512 encoded version)
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 15
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VINSERTI64x4 (EVEX.512 encoded version)
VL = 512
TEMP_DEST[VL-1:0] := SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] := SRC2[255:0]
    1: TMP_DEST[511:256] := SRC2[255:0]
ESAC.

FOR j := 0 TO 7
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VINSERTI128
TEMP[255:0] := SRC1[255:0]
CASE (imm8[0]) OF
  0: TEMP[127:0] := SRC2[127:0]
ESAC
DEST := TEMP

Intel C/C++ Compiler Intrinsic Equivalent
VINSERTI32x4 _mm512i_inserti32x4( __m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i_mask_inserti32x4(__m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i_maskz_inserti32x4( __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 _m256i_mm256_inserti32x4( __m256i a, __m128i b, int imm);
VINSERTI32x4 _m256i_mm256_mask_inserti32x4(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x4 _m256i_mm256_maskz_inserti32x4( __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x8 _m512i_mm512_inserti32x8( __m512i a, __m256i b, int imm);
VINSERTI32x8 _m512i_mm512_mask_inserti32x8(__m512i s, __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI32x8 _m512i_mm512_maskz_inserti32x8( __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI64x2 _m512i_mm512_inserti64x2( __m512i a, __m128i b, int imm);
VINSERTI64x2 _m512i_mm512_mask_inserti64x2(__m512i s, __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 _m512i_mm512_maskz_inserti64x2( __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x4 _m512i_mm512_inserti64x4( __m512i a, __m256i b, int imm);
VINSERTI64x4 _m512i_mm512_mask_inserti64x4(__m512i s, __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI64x4 _m512i_mm512_maskz_inserti64x4( __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI128 __m256i_mm256_insertf128_si256 ( __m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions
None

Other Exceptions
VEX-encoded instruction, see 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”. 
VMASKMOV—Conditional SIMD Packed Loads and Stores

**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.

---

**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>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
<tr>
<td>MVR</td>
<td>ModRM:r/m (w)</td>
<td>VEX.vvvv (r)</td>
<td>ModRMreg (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

---

**Opcode/Instruction Table**

<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 VMASKMOVPD 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 VMASKMOVPD 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>
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**

\[
\begin{align*}
\text{DEST}[31:0] & := \text{IF} \ (\text{SRC1}[31]) \ \text{Load}_32(\text{mem}) \ \text{ELSE} \ 0 \\
\text{DEST}[63:32] & := \text{IF} \ (\text{SRC1}[63]) \ \text{Load}_32(\text{mem} + 4) \ \text{ELSE} \ 0 \\
\text{DEST}[95:64] & := \text{IF} \ (\text{SRC1}[95]) \ \text{Load}_32(\text{mem} + 8) \ \text{ELSE} \ 0 \\
\text{DEST}[127:97] & := \text{IF} \ (\text{SRC1}[127]) \ \text{Load}_32(\text{mem} + 12) \ \text{ELSE} \ 0 \\
\text{DEST}[\text{MAXVL}-1:128] & := 0 \\
\end{align*}
\]

**VMASKMOVPS - 256-bit load**

\[
\begin{align*}
\text{DEST}[31:0] & := \text{IF} \ (\text{SRC1}[31]) \ \text{Load}_32(\text{mem}) \ \text{ELSE} \ 0 \\
\text{DEST}[63:32] & := \text{IF} \ (\text{SRC1}[63]) \ \text{Load}_32(\text{mem} + 4) \ \text{ELSE} \ 0 \\
\text{DEST}[95:64] & := \text{IF} \ (\text{SRC1}[95]) \ \text{Load}_32(\text{mem} + 8) \ \text{ELSE} \ 0 \\
\text{DEST}[127:96] & := \text{IF} \ (\text{SRC1}[127]) \ \text{Load}_32(\text{mem} + 16) \ \text{ELSE} \ 0 \\
\text{DEST}[191:160] & := \text{IF} \ (\text{SRC1}[191]) \ \text{Load}_32(\text{mem} + 20) \ \text{ELSE} \ 0 \\
\text{DEST}[223:192] & := \text{IF} \ (\text{SRC1}[223]) \ \text{Load}_32(\text{mem} + 24) \ \text{ELSE} \ 0 \\
\text{DEST}[255:224] & := \text{IF} \ (\text{SRC1}[255]) \ \text{Load}_32(\text{mem} + 28) \ \text{ELSE} \ 0 \\
\end{align*}
\]

**VMASKMOVPD - 128-bit load**

\[
\begin{align*}
\text{DEST}[63:0] & := \text{IF} \ (\text{SRC1}[63]) \ \text{Load}_64(\text{mem}) \ \text{ELSE} \ 0 \\
\text{DEST}[127:64] & := \text{IF} \ (\text{SRC1}[127]) \ \text{Load}_64(\text{mem} + 16) \ \text{ELSE} \ 0 \\
\text{DEST}[\text{MAXVL}-1:128] & := 0 \\
\end{align*}
\]

**VMASKMOVPD - 256-bit load**

\[
\begin{align*}
\text{DEST}[63:0] & := \text{IF} \ (\text{SRC1}[63]) \ \text{Load}_64(\text{mem}) \ \text{ELSE} \ 0 \\
\text{DEST}[127:64] & := \text{IF} \ (\text{SRC1}[127]) \ \text{Load}_64(\text{mem} + 16) \ \text{ELSE} \ 0 \\
\text{DEST}[195:128] & := \text{IF} \ (\text{SRC1}[191]) \ \text{Load}_64(\text{mem} + 24) \ \text{ELSE} \ 0 \\
\text{DEST}[255:196] & := \text{IF} \ (\text{SRC1}[255]) \ \text{Load}_64(\text{mem} + 28) \ \text{ELSE} \ 0 \\
\end{align*}
\]

**VMASKMOVPS - 128-bit store**

\[
\begin{align*}
\text{IF} \ (\text{SRC1}[31]) \ \text{DEST}[31:0] & := \text{SRC2}[31:0] \\
\text{IF} \ (\text{SRC1}[63]) \ \text{DEST}[63:32] & := \text{SRC2}[63:32] \\
\text{IF} \ (\text{SRC1}[95]) \ \text{DEST}[95:64] & := \text{SRC2}[95:64] \\
\text{IF} \ (\text{SRC1}[127]) \ \text{DEST}[127:96] & := \text{SRC2}[127:96] \\
\end{align*}
\]

**VMASKMOVPS - 256-bit store**

\[
\begin{align*}
\text{IF} \ (\text{SRC1}[31]) \ \text{DEST}[31:0] & := \text{SRC2}[31:0] \\
\text{IF} \ (\text{SRC1}[63]) \ \text{DEST}[63:32] & := \text{SRC2}[63:32] \\
\text{IF} \ (\text{SRC1}[95]) \ \text{DEST}[95:64] & := \text{SRC2}[95:64] \\
\text{IF} \ (\text{SRC1}[127]) \ \text{DEST}[127:96] & := \text{SRC2}[127:96] \\
\text{IF} \ (\text{SRC1}[159]) \ \text{DEST}[159:128] & := \text{SRC2}[159:128] \\
\text{IF} \ (\text{SRC1}[191]) \ \text{DEST}[191:160] & := \text{SRC2}[191:160] \\
\text{IF} \ (\text{SRC1}[223]) \ \text{DEST}[223:192] & := \text{SRC2}[223:192] \\
\text{IF} \ (\text{SRC1}[255]) \ \text{DEST}[255:224] & := \text{SRC2}[255:224] \\
\end{align*}
\]
VMASKMOV - 128-bit store
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]

VMASKMOV - 256-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
__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 Description</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 *);
VP2INTERSECTD void _mm512_2intersect_epi32(__m512i, __m512i, __mmask16 *, __mmask16 *);
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".
VPBLENDD — Blend Packed Dwords

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F3A.W0 02 /r ib VPBLENDD xmm1, xmm2, xmm3/m128, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Select dwords from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F3A.W0 02 /r ib VPBLENDD ymm1, ymm2, ymm3/m256, imm8</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX2</td>
<td>Select dwords from ymm2 and ymm3/m256 from mask specified in imm8 and store the values into ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RVMI</td>
<td>ModRM:reg (w)</td>
<td>VEX.vvvv (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

### 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 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 * 8
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SRC2[i+7:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN DEST[i+7:i] := SRC1[i+7:i]
    ELSE ; zeroing-masking
      DEST[i+7:i] := 0
    FI;
  FI;
ENDFOR

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

**VPBLENDMW (EVEX encoded versions)**

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

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SRC2[i+15:i]
  ELSE
    IF *merging-masking* ; merging-masking
      THEN DEST[i+15:i] := SRC1[i+15:i]
    ELSE ; zeroing-masking
      DEST[i+15:i] := 0
    FI;
  FI;
ENDFOR

DEST[MAXVL-1:VL] := 0

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

VPBLENDMB __m512i _mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i _mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i _mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);
VPBLENDMW __m512i _mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);
VPBLENDMW __m256i _mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);
VPBLENDMW __m128i _mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See 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

**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 7A /r VPBROADCASTB xmm1 (k1[1]), reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7A /r VPBROADCASTB ymm1 (k1[1]), reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7A /r VPBROADCASTB zmm1 (k1[1]), reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7B /r VPBROADCASTW xmm1 (k1[1]), reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7B /r VPBROADCASTW ymm1 (k1[1]), reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7B /r VPBROADCASTW zmm1 (k1[1]), reg</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 7C /r VPBROADCASTD xmm1 (k1[1]), r32</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all double-words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 (k1[1]), r32</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all double-words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 (k1[1]), r32</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Broadcast a 32-bit value from a GPR to all double-words in the 512-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 (k1[1]), r64</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quad-words in the 128-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 (k1[1]), r64</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512VL AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quad-words in the 256-bit destination subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 (k1[1]), r64</td>
<td>A</td>
<td>V/N.E.</td>
<td>AVX512F</td>
<td>Broadcast a 64-bit value from a GPR to all quad-words in the 512-bit destination subject to writemask k1.</td>
</tr>
</tbody>
</table>

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

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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>ModRMrm (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
  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);

Exceptions

EVEX-encoded instructions, see Table 2-55, "Type E7NM Class Exception Conditions"; additionally:

#UD If EVEX.vvvv != 1111B.
<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<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 VPBROADCASTB xmm1, xmm2/m8</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 VPBROADCASTB ymm1, xmm2/m8</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 VPBROADCASTB xmm1{k1}{z}, xmm2/m8</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 VPBROADCASTB ymm1{k1}{z}, xmm2/m8</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 VPBROADCASTB zmm1{k1}{z}, xmm2/m8</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 VPBROADCASTW xmm1, xmm2/m16</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 VPBROADCASTW ymm1, xmm2/m16</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 VPBROADCASTW xmm1{k1}{z}, xmm2/m16</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 VPBROADCASTW ymm1{k1}{z}, xmm2/m16</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 VPBROADCASTW zmm1{k1}{z}, xmm2/m16</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 VPBROADCASTD xmm1, xmm2/m32</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 VPBROADCASTD ymm1, xmm2/m32</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 VPBROADCASTD xmm1 {k1}{z}, xmm2/m32</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 VPBROADCASTD ymm1 {k1}{z}, xmm2/m32</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 VPBROADCASTD zmm1 {k1}{z}, xmm2/m32</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 VPBROADCASTQ xmm1, xmm2/m64</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 VPBROADCASTQ ymm1, xmm2/m64</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 VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64</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 VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64</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 VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64</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 VPBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64</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>
</tbody>
</table>
Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTI32x2 ymm1 (k1)[z], xmm2/m64: Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.

VEX512-encoded VPBROADCASTI32x2 zmm1 (k1)[z], xmm2/m64: Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.

VXP256-encoded VPBROADCASTI128 ymm1, m128: Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.

EVEX-encoded VPBROADCASTI128 ymm1 {k1}{z}, xmm2/m64: Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1.

EVEX-encoded VPBROADCASTI128 zmm1 {k1}{z}, xmm2/m64: Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.

EVEX-encoded VPBROADCASTI32X4 ymm1 {k1}{z}, m128: Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1.

EVEX-encoded VPBROADCASTI32X4 zmm1 {k1}{z}, m128: Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.

EVEX-encoded VPBROADCASTI64X2 zmm1 {k1}{z}, m256: Broadcast 256 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.

EVEX-encoded VPBROADCASTI64X4 zmm1 {k1}[z], m256: Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.

EVEX-encoded VPBROADCASTI32X8 zmm1 {k1}{z}, m256: Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1.

EVEX-encoded VPBROADCASTI64X4 zmm1 [k1][z], m256: Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.

Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register.

VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VPBROADCASTI32X4 and VPBROADCASTI64X4 are reserved and will #UD.
Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD. If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

Figure 5-17. VPBROADCASTD Operation (128-bit version)

Figure 5-18. VPBROADCASTQ Operation (256-bit version)
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* ; zeroing-masking
    ELSE
      DEST[i+7:i] := 0
    FI
  FI
ENDFOR

DEST[MAXVL-1:VL] := 0

---

**Figure 5-19. VBROADCASTI128 Operation (256-bit version)**

**Figure 5-20. VBROADCASTI256 Operation (512-bit version)**
VPBROADCAST (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

VPBROADCAST (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

VPBROADCAST (VEX.256 encoded version)
temp := SRC[31:0]
DEST[31:0] := temp
DEST[63:32] := temp
DEST[95:64] := temp
DEST[127:96] := temp
DEST[159:128] := temp
DEST[191:160] := temp
DEST[223:192] := temp
DEST[255:224] := temp
DEST[MAXVL-1:256] := 0

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

VPBROADCAST—Load Integer and Broadcast
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)

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

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

VBROADCASTI32x2 (EVEX encoded versions)

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

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

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 \(KL-1\)
  \[i := j \times 32\]
  \[n := (j \mod 4) \times 32\]
  IF \(k1[j] \text{ OR } \text{no writemask}\)
    THEN \(\text{DEST}[i+31:i] := \text{SRC}[n+31:n]\)
    ELSE
      IF \(*\text{merging-masking}\) ; merging-masking
        THEN \(*\text{DEST}[i+31:i]\) remains unchanged*
        ELSE ; zeroing-masking
          \(\text{DEST}[i+31:i] := 0\)
      FI
  FI;
ENDFOR

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

**VBROADCASTI64X2 (EVEX encoded versions)**

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

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

**VBROADCASTI32X8 (EVEX.U1.512 encoded version)**

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

\(\text{DEST}[\text{MAXVL-1:VL}] := 0\)
VBROADCASTI64X4 (EVEX.512 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCAST _m512i_mm512_broadcastb_epi8(__m128i a);
VPBROADCAST _m512i_mm512_mask_broadcastb_epi8(__m128i s, __mmask64 k, __m128i a);
VPBROADCAST _m512i_mm512_maskz_broadcastb_epi8(__mmask64 k, __m128i a);
VPBROADCAST _m256i_mm256_broadcastb_epi8(__m128i a);
VPBROADCAST _m256i_mm256_mask_broadcastb_epi8(__m256i s, __mmask32 k, __m128i a);
VPBROADCAST _m256i_mm256_maskz_broadcastb_epi8(__mmask32 k, __m128i a);
VPBROADCAST _m128i_mm128_broadcastb_epi8(__m128i a);
VPBROADCAST _m128i_mm128_mask_broadcastb_epi8(__m128i s, __mmask16 k, __m128i a);
VPBROADCAST _m128i_mm128_maskz_broadcastb_epi8(__mmask16 k, __m128i a);
VPBROADCAST _m512i_mm512_broadcastd_epi32(__m128i a);
VPBROADCAST _m512i_mm512_mask_broadcastd_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCAST _m512i_mm512_maskz_broadcastd_epi32(__mmask16 k, __m128i a);
VPBROADCAST _m256i_mm256_broadcastd_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCAST _m256i_mm256_mask_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCAST _m256i_mm256_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCAST _m128i_mm128_broadcastd_epi32(__m128i a);
VPBROADCAST _m128i_mm128_mask_broadcastd_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCAST _m128i_mm128_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCAST _m512i_mm512_broadcastq_epi64(__m128i a);
VPBROADCAST _m512i_mm512_mask_broadcastq_epi64(__m512i s, __mmask8 k, __m128i a);
VPBROADCAST _m512i_mm512_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCAST _m256i_mm256_broadcastq_epi64(__m256i s, __mmask8 k, __m128i a);
VPBROADCAST _m256i_mm256_mask_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCAST _m256i_mm256_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCAST _m128i_mm128_broadcastq_epi64(__m128i a);
VPBROADCAST _m128i_mm128_mask_broadcastq_epi64(__m128i s, __mmask8 k, __m128i a);
VPBROADCAST _m128i_mm128_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCAST _m512i_mm512_broadcastw_epi16(__m128i a);
VPBROADCAST _m512i_mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCAST _m512i_mm512_maskz_broadcastw_epi16(__mmask32 k, __m128i a);
VPBROADCAST _m256i_mm256_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCAST _m256i_mm256_mask_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCAST _m256i_mm256_maskz_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCAST _m128i_mm128_broadcastw_epi16(__m128i a);
VPBROADCAST _m128i_mm128_mask_broadcastw_epi16(__m128i s, __mmask16 k, __m128i a);
VPBROADCAST _m128i_mm128_maskz_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCAST _m512i_mm512_broadcastq_epi32(__m128i a);
VPBROADCAST _m512i_mm512_mask_broadcastq_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCAST _m512i_mm512_maskz_broadcastq_epi32(__mmask16 k, __m128i a);
VPBROADCAST _m256i_mm256_broadcastq_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCAST _m256i_mm256_mask_broadcastq_epi32(__mmask8 k, __m128i a);
VPBROADCAST _m256i_mm256_maskz_broadcastq_epi32(__mmask8 k, __m128i a);
VPBROADCAST _m128i_mm128_broadcastq_epi32(__m128i a);
VPBROADCAST _m128i_mm128_mask_broadcastq_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCAST _m128i_mm128_maskz_broadcastq_epi32(__mmask8 k, __m128i a);
VPBROADCAST _m512i_mm512_broadcastw_epi16(__m128i a);
VPBROADCAST _m512i_mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCAST _m512i_mm512_maskz_broadcastw_epi16(__mmask32 k, __m128i a);
VPBROADCAST _m256i_mm256_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCAST _m256i_mm256_mask_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCAST _m256i_mm256_maskz_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCAST _m128i_mm128_broadcastw_epi16(__m128i a);
VPBROADCAST _m128i_mm128_mask_broadcastw_epi16(__m128i s, __mmask16 k, __m128i a);
VPBROADCAST _m128i_mm128_maskz_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCAST _m512i_mm512_broadcastq_epi32(__m128i a);
VPBROADCAST _m512i_mm512_mask_broadcastq_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCAST _m512i_mm512_maskz_broadcastq_epi32(__mmask16 k, __m128i a);
VPBROADCAST _m256i_mm256_broadcastq_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCAST _m256i_mm256_maskroadcastq_epi32(__mmask8 k, __m128i a);
VPBROADCAST _m256i_mm256_maskz_broadcastq_epi32(__mmask8 k, __m128i a);
VPBROADCAST _m128i_mm128_broadcastq_epi32(__m128i a);
VPBROADCAST _m128i_mm128_mask_broadcastq_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCAST _m128i_mm128_maskz_broadcastq_epi32(__mmask8 k, __m128i 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

### 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.F3.0F38.W1 2A /r VPBROADCASTMB2Q xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low byte value in k1 to two locations in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 2A /r VPBROADCASTMB2Q ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low byte value in k1 to four locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 2A /r VPBROADCASTMB2Q zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Broadcast low byte value in k1 to eight locations in zmm1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 3A /r VPBROADCASTMW2D xmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low word value in k1 to four locations in xmm1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 3A /r VPBROADCASTMW2D ymm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512VL AVX512CD</td>
<td>Broadcast low word value in k1 to eight locations in ymm1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 3A /r VPBROADCASTMW2D zmm1, k1</td>
<td>RM</td>
<td>V/V</td>
<td>AVX512CD</td>
<td>Broadcast low word value in k1 to sixteen locations in zmm1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

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

#### Description

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

#### Operation

**VPBROADCASTMB2Q**

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

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

\[ i := j \times 64 \]

\[ \text{DEST}[i+63:i] := \text{ZeroExtend} \left( \text{SRC}[7:0] \right) \]

ENDFOR

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

**VPBROADCASTMW2D**

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

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

\[ i := j \times 32 \]

\[ \text{DEST}[i+31:i] := \text{ZeroExtend} \left( \text{SRC}[15:0] \right) \]

ENDFOR

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

**Op/En/Instruction**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 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>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>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>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>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>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>
</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/r/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 5-8. Pseudo-Op and VPCMP* Implementation

<table>
<thead>
<tr>
<th>Pseudo-Op</th>
<th>PCMPM Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPCMPEQ* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 0</td>
</tr>
<tr>
<td>VPCMPLT* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 1</td>
</tr>
<tr>
<td>VPCMPLE* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 2</td>
</tr>
<tr>
<td>VPCMPNEQ* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 4</td>
</tr>
<tr>
<td>VPPCMPLT* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 5</td>
</tr>
<tr>
<td>VPCMPNLE* reg1, reg2, reg3</td>
<td>VPCMP* reg1, reg2, reg3, 6</td>
</tr>
</tbody>
</table>

### Operation

CASE (COMPARISON PREDICATE) OF

0: OP := EQ;
1: OP := LT;
2: OP := LE;
3: OP := FALSE;
4: OP := NEQ;
5: OP := NLT;
6: OP := NLE;
7: OP := TRUE;

ESAC;

**VPCMPB (EVEX encoded versions)**

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

FOR j := 0 TO KL-1
    i := j * 8
    IF k2[j] OR *no writemask*
        THEN
            CMP := SRC1[i+7:i] OP SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
        ELSE DEST[j] = 0 ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] := 0
VPCMPUB (EVEX encoded versions)

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

FOR j := 0 TO KL-1
  i := j * 8
  IF k2[j] OR *no writemask*
    THEN
      CMP := SRC1[i+7:i] OP SRC2[i+7:i];
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
    ELSE DEST[j] = 0 ; 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_cmp_epu8_mask( __m512i a, __m512i b, int cmp);
VPCMPB __mmask32 _mm256_cmp_epi8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __mmask32 _mm256_cmp_epu8_mask( __m256i a, __m256i b, int cmp);
VPCMPB __mmask16 _mm_cmp_epi8_mask( __m128i a, __m128i b, int cmp);
VPCMPB __mmask16 _mm_cmp_epu8_mask( __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”. 
VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask

### 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>
<th>Description</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>
<td></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;
ENDIF
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;
ENDIF
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);
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);
VPCMPUD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions”.

VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask Vol. 2C 5-327
VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask

<table>
<thead>
<tr>
<th>Opcode/Opcode Instruction</th>
<th>Op/Op En</th>
<th>64/32 64/32 64/32 64/32 64/32 64/32 64/32 bit Mode bit Mode bit Mode bit Mode bit Mode bit Mode Support Support Support Support Support Support</th>
<th>CPUID Feature Feature Feature Feature Feature Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 1F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>AVX512F</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 1F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>AVX512F</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 1F /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 1E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>AVX512F</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 1E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>AVX512F</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 1E /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Operand Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op/En</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-8.
**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;

**VPCMPQ (EVEX encoded versions)**

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

FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
                FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
            ELSE DEST[j] := 0 ; zeroing-masking only
        FI;
    ENDFOR

DEST[MAX_KL-1:KL] := 0

**VPCMPUQ (EVEX encoded versions)**

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

FOR j := 0 TO KL-1
    i := j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP := SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];
                FI;
            IF CMP = TRUE
                THEN DEST[j] := 1;
                ELSE DEST[j] := 0; FI;
            ELSE DEST[j] := 0 ; zeroing-masking only
        FI;
    ENDFOR

DEST[MAX_KL-1:KL] := 0
INSTRUCTION SET REFERENCE, V-Z

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|lt|neq]_epi64_mask( __m512i a, __m512i b);
VPCMPQ __mmask8 _mm512_mask_cmp[eq|gt|lt|neq]_epi64_mask( __mmask8 k, __m512i a, __m512i b);
VPCMPQ __mmask8 _mm512_cmp_epi64_mask( __m128i a, __m128i b);
VPCMPQ __mmask8 _mm_mask_cmp_epi64_mask( __mmask8 k, __m128i a, __m128i b);
VPCMPQ __mmask8 _mm_cmp_epi64_mask( __m128i a, __m128i b);
VPCMPQ __mmask8 _mm_mask_cmp_epi64_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
## VPCMPW/VPCMPUW—Compare Packed Word 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.W1 3F /r ib VPCMPW k1 [k2], xmm2, xmm3/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 3F /r ib VPCMPW k1 [k2], ymm2, ymm3/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed signed word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 3F /r ib VPCMPW k1 [k2], zmm2, zmm3/m512, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed signed word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 3E /r ib VPCMPUW k1 [k2], xmm2, xmm3/m128, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 3E /r ib VPCMPUW k1 [k2], ymm2, ymm3/m256, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</td>
</tr>
<tr>
<td>VPCMPUW k1 [k2], zmm2, zmm3/m512, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512BW</td>
<td>Compare packed unsigned word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.</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>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;
  ENDIF
  ENDFOR
  DEST[MAX_KL-1:KL] := 0

VPCMPUW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KL-1
  i := j * 16
  IF k2[j] OR *no writemask*
    THEN
      CMP := SRC1[i+15:i] OP SRC2[i+15:i];
      IF CMP = TRUE
        THEN DEST[j] := 1;
        ELSE DEST[j] := 0; FI;
      ELSE DEST[j] = 0 ; zeroing-masking only
    FI;
  ENDIF
  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 __mmask8 _mm_mask_cmp_epi16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m512i a, __m512i b);
VPCMPW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m256i a, __m256i b);
VPCMPW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m128i a, __m128i b);
VPCMPW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask8 m, __m128i a, __m128i b);
VPCMPUW __mmask32 _mm512_cmp_epi16_mask(__m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp_epi16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp_epi16_mask(__m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp_epi16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp_epi16_mask(__m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp_epi16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPUW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb 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>

**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 \[KL-1\]:

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

\[\text{DEST.byte}\[k\] := SRC.byte]\[j\]

\[k := k + 1\]

**VPCOMPRESSB reg-reg form**

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

\[k := 0\]

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

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

\[\text{DEST.byte}\[k\] := SRC.byte]\[j\]

\[k := k + 1\]

IF *merging-masking*:

\[*\text{DEST[VL-1:k*8]} \text{ remains unchanged}\]*

ELSE \[\text{DEST[VL-1:k*8]} := 0\]

\[\text{DEST[MAX}_V\text{L-1:VL]} := 0\]

**VPCOMPRESSW store form**

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

\[k := 0\]

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

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

\[\text{DEST.word}\[k\] := SRC.word]\[j\]

\[k := k + 1\]

**VPCOMPRESSW reg-reg form**

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

\[k := 0\]

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

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

\[\text{DEST.word}\[k\] := SRC.word]\[j\]

\[k := k + 1\]

IF *merging-masking*:

\[*\text{DEST[VL-1:k*16]} \text{ remains unchanged}\]*

ELSE \[\text{DEST[VL-1:k*16]} := 0\]

\[\text{DEST[MAX}_V\text{L-1:VL]} := 0\]
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

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<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)

SIZE := 32

k := 0

FOR j := 0 TO KL-1

i := j * 32

IF (k1[j]) OR *no controlmask* THEN

DEST[k+SIZE-1:k] := SRC[i+31:i]

k := k + SIZE

FI;

ENDFOR;
VPCOMPRESSD (EVEX encoded versions) reg-reg form

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

Intel C/C++ Compiler Intrinsic Equivalent
VPCOMPRESSD __m512i _mm512_mask_compress_epi32(__m512i s, __mmask16 c, __m512i a);
VPCOMPRESSD __m512i _mm512_maskz_compress_epi32( __mmask16 c, __m512i a);
VPCOMPRESSD void _mm512_mask_compressstoreu_epi32(void * a, __mmask16 c, __m512i s);
VPCOMPRESSD __m256i _mm256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i _mm256_maskz_compress_epi32( __mmask8 c, __m256i a);
VPCOMPRESSD void _mm256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i _mm_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i _mm_maskz_compress_epi32( __mmask8 c, __m128i a);
VPCOMPRESSD void _mm_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”.

INSTRUCTION SET REFERENCE, V-Z
VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register

Op/En Instruction | Op/En | 64/32 bit Mode Support | CPUID Feature Flag | Description
--- | --- | --- | --- | ---
EVEX.128.66.0F38.W1 8B /r VPCOMPRESSQ xmm1/m128 {k1}{z}, xmm2 | A | V/V | AVX512VL AVX512F | Compress packed quadword integer values from xmm2 to xmm1/m128 using controlmask k1.
EVEX.256.66.0F38.W1 8B /r VPCOMPRESSQ ymm1/m256 {k1}{z}, ymm2 | A | V/V | AVX512VL AVX512F | Compress packed quadword integer values from ymm2 to ymm1/m256 using controlmask k1.
EVEX.512.66.0F38.W1 8B /r VPCOMPRESSQ zmm1/m512 {k1}{z}, zmm2 | A | V/V | AVX512F | Compress packed quadword integer values from zmm2 to zmm1/m512 using controlmask k1.

Instruction Operand Encoding

<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".
# VPCONFLICTD/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 VPCONFLICTD 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 VPCONFLICTD 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 VPCONFLICTD 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.W0 C4 /r VPCONFLICTQ 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.W0 C4 /r VPCONFLICTQ 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.W0 C4 /r VPCONFLICTQ 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 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.

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

**VPCONFlicTD**

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

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

DEST[MAXVL-1:VL] := 0

**VPCONFlicTQ**

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

FOR j := 0 TO KL-1
  i := j*64
  IF MaskBit(j) OR *no writemask* THEN
    FOR k := 0 TO j-1
      m := k*64
      IF (SRC[i+63:i] = SRC[m+63:m]) THEN
        DEST[i+k] := 1
      ELSE
        DEST[i+k] := 0
      FI
    ENDFOR
    DEST[i+63:i+j] := 0
  ELSE
    IF *merging-masking* THEN
      *DEST[i+63:i] remains unchanged*
    ELSE
      DEST[i+63:i] := 0
    FI
  FI
ENDFOR

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

VPCONFLICTD __m512i _mm512_conflict_epi32(__m512i a);
VPCONFLICTD __m512i _mm512_mask_conflict_epi32(__m512i s, __mmask16 m, __m512i a);
VPCONFLICTD __m512i _mm512_maskz_conflict_epi32(__mmask16 m, __m512i a);
VPCONFLICTQ __m512i _mm512_conflict_epi64(__m512i a);
VPCONFLICTQ __m512i _mm512_mask_conflict_epi64(__m512i s, __mmask8 m, __m512i a);
VPCONFLICTQ __m512i _mm512_maskz_conflict_epi64(__mmask8 m, __m512i a);
VPCONFLICTD __m256i _mm256_conflict_epi32(__m256i a);
VPCONFLICTD __m256i _mm256_mask_conflict_epi32(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTD __m256i _mm256_maskz_conflict_epi32(__mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_conflict_epi64(__m256i a);
VPCONFLICTQ __m256i _mm256_mask_conflict_epi64(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_maskz_conflict_epi64(__mmask8 m, __m256i a);
VPCONFLICTD __m128i _mm_conflict_epi32(__m128i a);
VPCONFLICTD __m128i _mm_mask_conflict_epi32(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTD __m128i _mm_maskz_conflict_epi32(__mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_conflict_epi64(__m128i a);
VPCONFLICTQ __m128i _mm_mask_conflict_epi64(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_maskz_conflict_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions”. 
### 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<a href="z">k1</a>, 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<a href="z">k1</a>, 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<a href="z">k1</a>, 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.
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]
    p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
    p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
    p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
    p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) * SIGN_EXTEND(t.byte[3])
    DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word
  ELSE IF *zeroing*:
    DEST.dword[i] := 0
  ELSE: // Merge masking, dest element unchanged
    DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPDPBUSD __m128i _mm_dpbusd_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, __m256i);
VPDPBUSD __m256i _mm256_dpbusd_epi32(__m256i, __m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_mask_dpbusd_epi32(__m256i, __mmask8, __m256i, __m256i, __m256i);
VPDPBUSD __m256i _mm256_maskz_dpbusd_epi32(__mmask8, __m256i, __m256i, __m256i, __m256i);
VPDPBUSD __m512i _mm512_dpbusd_epi32(__m512i, __m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_mask_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i, __m512i);
VPDPBUSD __m512i _mm512_maskz_dpbusd_epi32(__mmask16, __m512i, __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 AVX512VL</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 AVX512VL</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:reg/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:reg/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]
    ENDIF
  ELSE:
    // Merge masking, dest element unchanged
    DEST.dword[i] := ORIGDEST.dword[i]
  ENDIF
  ELSE IF *zeroing*:
    DEST.dword[i] := 0
  ELSE:
    DEST.dword[i] := ORIGDEST.dword[i]
  ENDIF
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, __mmask8, __m128i, __m128i);
VPDPBUSDS _m128i _mm_maskz_dpbusds_epi32(__mmask8, __m128i, __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(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSDS _m256i _mm256_maskz_dpbusds_epi32(__mmask16, __m256i, __m256i, __m256i);
VPDPBUSDS _m512i _mm512_dpbusds_epi32(__m512i, __m512i, __m512i);
VPDPBUSDS _m512i _mm512_mask_dpbusds_epi32(__mmask16, __m512i, __m512i, __m512i);
VPDPBUSDS _m512i _mm512_maskz_dpbusds_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”.

**VPDPWSSD — Multiply and Add Signed Word 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>VEX.128.66.0F38.W0 52 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 2 pairs signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 52 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 2 pairs signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 52 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 2 pairs signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 52 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 2 pairs signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 52 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI</td>
<td>Multiply groups of 2 pairs signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, under writemask k1.</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>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**

**VPDPWSSD 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] := ORIGDEST.dword[i] + p1dword + p2dword
    DEST[MAX_VL-1:VL] := 0
VPDPWSSD 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] := 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

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<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 53 /r VPDPWSSDS xmm1, xmm2, xmm3/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 2 pairs of signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation.</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1, ymm2, ymm3/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX-VNNI</td>
<td>Multiply groups of 2 pairs of signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 2 pairs of signed words in xmm3/m128/m32bcst with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI AVX512VL</td>
<td>Multiply groups of 2 pairs of signed words in ymm3/m256/m32bcst with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation, under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 53 /r VPDPWSSDS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VNNI</td>
<td>Multiply groups of 2 pairs of signed words in zmm3/m512/m32bcst with corresponding signed words of zmm2, summing those products and adding them to doubleword result in zmm1, with signed saturation, under writemask k1.</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 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 \(KL-1\):

\[p1dword := \text{SIGN EXTEND}(\text{SRC1.word}[2*i+0]) \times \text{SIGN EXTEND}(\text{SRC2.word}[2*i+0])\]

\[p2dword := \text{SIGN EXTEND}(\text{SRC1.word}[2*i+1]) \times \text{SIGN EXTEND}(\text{SRC2.word}[2*i+1])\]

\(\text{DEST.dword}[i] := \text{SIGNED_DWORD_SATURATE}(\text{ORIGDEST.dword}[i] + p1dword + p2dword)\)

\(\text{DEST}[\text{MAX}_V\text{L}-1:VL] := 0\)

**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:

\[t := \text{SRC2.dword}[0]\]

ELSE:

\[t := \text{SRC2.dword}[i]\]

\[p1dword := \text{SIGN EXTEND}(\text{SRC1.word}[2*i]) \times \text{SIGN EXTEND}(t.word[0])\]

\[p2dword := \text{SIGN EXTEND}(\text{SRC1.word}[2*i+1]) \times \text{SIGN EXTEND}(t.word[1])\]

\(\text{DEST.dword}[i] := \text{SIGNED_DWORD_SATURATE}(\text{ORIGDEST.dword}[i] + p1dword + p2dword)\)

ELSE IF *zeroing*:

\(\text{DEST.dword}[i] := 0\)

ELSE: // Merge masking, dest element unchanged

\(\text{DEST.dword}[i] := \text{ORIGDEST.dword}[i]\)

\(\text{DEST}[\text{MAX}_V\text{L}-1:VL] := 0\)

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

- \(\text{VPDPWSSDS} \_\_m128i \_\_mm_dpwssds\_avx\_epi32(\_\_m128i, \_\_m128i, \_\_m128i);\)
- \(\text{VPDPWSSDS} \_\_m128i \_\_mm_dpwssds\_epi32(\_\_m128i, \_\_m128i, \_\_m128i);\)
- \(\text{VPDPWSSDS} \_\_m128i \_\_mm_mask_dpwssds\_epi32(\_\_m128i, \_\_mmask8, \_\_m128i, \_\_m128i);\)
- \(\text{VPDPWSSDS} \_\_m128i \_\_mm_maskz_dpwssd\_epi32(\_\_mmask8, \_\_m128i, \_\_m128i, \_\_m128i);\)
- \(\text{VPDPWSSDS} \_\_m256i \_\_mm256\_dpwssds\_avx\_epi32(\_\_m256i, \_\_m256i, \_\_m256i);\)
- \(\text{VPDPWSSDS} \_\_m256i \_\_mm256\_dpwssds\_epi32(\_\_m256i, \_\_mmask8, \_\_m256i, \_\_m256i);\)
- \(\text{VPDPWSSDS} \_\_m256i \_\_mm256\_mask_dpwssds\_epi32(\_\_mmask8, \_\_m256i, \_\_m256i, \_\_m256i);\)
- \(\text{VPDPWSSDS} \_\_m256i \_\_mm256\_maskz\_dpwssd\_epi32(\_\_mmask8, \_\_m256i, \_\_m256i, \_\_m256i);\)
- \(\text{VPDPWSSDS} \_\_m512i \_\_mm512\_dpwssds\_epi32(\_\_mm512i, \_\_mm512i, \_\_mm512i);\)
- \(\text{VPDPWSSDS} \_\_m512i \_\_mm512\_mask\_dpwssds\_epi32(\_\_mmask16, \_\_m512i, \_\_m512i, \_\_m512i);\)
- \(\text{VPDPWSSDS} \_\_m512i \_\_mm512\_maskz\_dpwssd\_epi32(\_\_mmask16, \_\_m512i, \_\_m512i, \_\_m512i);\)

**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

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<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 06/r ib</td>
<td>RVMI</td>
<td>V/V</td>
<td>AVX</td>
<td>Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1.</td>
</tr>
</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 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)

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

**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)**

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

\[
\begin{align*}
&IF \ VL = 128: \\
&\quad n := 3; \\
&ELSE IF \ VL = 256: \\
&\quad n := 4; \\
&ELSE IF \ VL = 512: \\
&\quad n := 5; \\
&FI; \\
&FOR j := 0 TO KL-1: \\
&\quad id := SRC1[j*8 + n : j*8]; // location of the source byte \\
&\quad IF k1[j] OR *no writemask* THEN \\
&\quad \quad DEST[j*8 + 7:j*8] := SRC2[id*8 + 7: id*8]; \\
&\quad ELSE IF zeroing-masking THEN \\
&\quad \quad DEST[j*8 + 7:j*8] := 0; \\
&\quad *ELSE \\
&\quad \quad DEST[j*8 + 7:j*8] \text{ remains unchanged} \\
&\quad FI \\
&ENDFOR \\
&DEST[MAX_VL-1:VL] := 0;
\]

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

VPERMB __m512i _mm512_permutexvar_epi8(__m512i idx, __m512i a);

---

**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>

---

**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 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>
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 __mm128_permutexvar_epi8(__m128i idx, __m128i a);
VPERMB __m128i __mm128_mask_permutexvar_epi8(__m128i s, __mmask16 k, __m128i idx, __m128i a);
VPERMB __m128i __mm128_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—Permuted 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 V/V</td>
<td>AVX2</td>
<td>Permutes doublewords in ymm3/m256 using indices in ymm2 and stores the result in ymm1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w0 36 /r VPERMD ymm1 [k1]{z}, ymm2, ymm3/m256/m32bcst</td>
<td>B V/V</td>
<td>AVX512VL AVX512F</td>
<td>Permutes doublewords in ymm3/m256/m32bcst using indices in ymm2 and stores the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 36 /r VPERMD zmm1 [k1]{z}, zmm2, zmm3/m512/m32bcst</td>
<td>B V/V</td>
<td>AVX512F</td>
<td>Permutes doublewords in zmm3/m512/m32bcst using indices in zmm2 and stores the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w1 8D /r VPERMW xmm1 [k1]{z}, xmm2, xmm3/m128</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permutes word integers in xmm3/m128 using indexes in xmm2 and stores the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 8D /r VPERMW ymm1 [k1]{z}, ymm2, ymm3/m256</td>
<td>C V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Permutes word integers in ymm3/m256 using indexes in ymm2 and stores the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w1 8D /r VPERMW zmm1 [k1]{z}, zmm2, zmm3/m512</td>
<td>C V/V</td>
<td>AVX512BW</td>
<td>Permutes word integers in zmm3/m512 using indexes in zmm2 and stores 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>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

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

ENDFOR

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

VPERMD __m512i __mm512_permutexvar_epi32(__m512i idx, __m512i a);
VPERMD __m512i __mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i __mm512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i __mm256_permutexvar_epi32(__m256i idx, __m256i a);
VPERMD __m256i __mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i __mm256_maskz_permutexvar_epi32(__mmask8 k, __m256i idx, __m256i a);
VPERMD __m128i __mm_permutexvar_epi32(__m128i idx, __m128i a);
VPERMD __m128i __mm_mask_permutexvar_epi32(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMD __m128i __mm_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.
VPERMIB2B—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 VPERMIB2B 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 VPERMIB2B 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 VPERMIB2B 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

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

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

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

Bits (MAX_VL-1:256/128) of the destination are zeroed for VL=256,128.
Operation

VPERMI2B (EVEX encoded versions)

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

IF VL = 128:
  id := 3;
ELSE IF VL = 256:
  id := 4;
ELSE IF VL = 512:
  id := 5;
FI;

TMP_DEST[VL-1:0] := DEST[VL-1:0];
FOR j := 0 TO KL-1
  off := 8*SRC1[j*8 + id: j*8];
  IF k1[j] OR *no writemask*:
    DEST[j*8 + 7: j*8] := TMP_DEST[j*8+id+1]? SRC2[off+7:off]: SRC1[off+7:off];
  ELSE IF *zeroing-masking*
    DEST[j*8 + 7: j*8] := 0;
  ELSE
    DEST[j*8 + 7: j*8] remains unchanged*
  FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;

Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2B __m512i _mm512_permutex2var_epi8(__m512i a, __m512i idx, __m512i b);
VPERMI2B __m512i _mm512_mask2_permutex2var_epi8(__m512i a, __m512i idx, __mmask64 k, __m512i b);
VPERMI2B __m512i _mm512_maskz_permutex2var_epi8(__mmask64 k, __m512i a, __m512i idx, __m512i b);
VPERMI2B __m256i _mm256_permutex2var_epi8(__m256i a, __m256i idx, __m256i b);
VPERMI2B __m256i _mm256_mask2_permutex2var_epi8(__m256i a, __m256i idx, __mmask32 k, __m256i b);
VPERMI2B __m256i _mm256_maskz_permutex2var_epi8(__mmask32 k, __m256i a, __m256i idx, __m256i b);
VPERMI2B __m128i _mm_permutex2var_epi8(__m128i a, __m128i idx, __m128i b);
VPERMI2B __m128i _mm_mask2_permutex2var_epi8(__m128i a, __m128i idx, __mmask16 k, __m128i b);
VPERMI2B __m128i _mm_maskz_permutex2var_epi8(__mmask16 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type E4NF.nb 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 VPERM12W 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 xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 75 /r VPERM12W 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 ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 75 /r VPERM12W 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 zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 76 /r VPERM12D 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 xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 76 /r VPERM12D 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 ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 76 /r VPERM12D 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 zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 76 /r VPERM12Q 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 xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 76 /r VPERM12Q 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 ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 76 /r VPERM12Q 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 zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 77 /r VPERM12PS 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 xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 77 /r VPERM12PS 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 ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 77 /r VPERM12PS 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 zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.</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.

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<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 77 /r VPERMI2PD 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 xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 77 /r VPERMI2PD 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 ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 77 /r VPERMI2PD 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 zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.</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</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg</td>
<td>EVEX.vvvv</td>
<td>ModRM:r/m</td>
<td>NA</td>
</tr>
</tbody>
</table>
Operation

**VPERMI2W (EVEX encoded versions)**

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

IF VL = 128
   id := 2
FI;
IF VL = 256
   id := 3
FI;
IF VL = 512
   id := 4
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
   i := j * 16
   off := 16*TMP_DEST[i+id]
   IF k1[j] OR *no writemask*
      THEN
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+15:i] := 0
         FI
      FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

**VPERMI2D/VPERMI2PS (EVEX encoded versions)**

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

IF VL = 128
   id := 1
FI;
IF VL = 256
   id := 2
FI;
IF VL = 512
   id := 3
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
   i := j * 32
   off := 32*TMP_DEST[i+id]
   IF k1[j] OR *no writemask*
      THEN
         IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
            ELSE
      ELSE
VPERM2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

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

VPERM2Q/VPERM2PD (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
        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;
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);
VPERMI2D __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 _mm128_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMI2D __m128i _mm128_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2D __m128i _mm128_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2D __m128i _mm128_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 _mm128_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm128_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm128_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMI2PD __m128d _mm128_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 _mm128_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm128_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm128_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMI2PS __m128 _mm128_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMI2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMI2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMI2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERMI2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMI2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
VPERMI2D/Q/PS/PD: See Table 2-50, “Type E4NF Class Exception Conditions”.
VPERMI2W: See Exceptions Type E4NF.nb in Table 2-50, “Type E4NF Class Exception Conditions”.
### VPERMILPD—Permutes 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.

![Figure 5-23. VPERMILPD Operation](image)

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

![Figure 5-24. VPERMILPD Shuffle Control](image)

(immediate control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.
Operation

VPERMILPD (EVEX immediate versions)

(KL, VL) = (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+63:i] := SRC1[63:0];
        ELSE TMP_SRC1[i+63:i] := SRC1[i+63:i];
    FI;
ENDFOR;
IF (imm8[0] = 0) THEN TMP_DEST[63:0] := SRC1[63:0]; FI;
IF (imm8[0] = 1) THEN TMP_DEST[63:0] := TMP_SRC1[127:64]; FI;
IF (imm8[1] = 0) THEN TMP_DEST[127:64] := TMP_SRC1[63:0]; FI;
IF VL >= 256
    FI;
    IF VL >= 512
    FI;
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMILPD (256-bit immediate version)
IF (imm8[0] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] := SRC1[63:0]
DEST[MAXVL-1:256] := 0
**VPERMILPD (128-bit immediate version)**

IF (imm8[0] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] := SRC1[63:0]
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 VL >= 256
FI;
IF VL >= 512
    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*
                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);
VPERMILPD __m128d _mm128_maskz_permute_pd(__mmask8 k, __m128d 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.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]);
    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
VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

```
F1;
F1;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMILPS (256-bit variable version)
DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] := Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:256] := 0

VPERMILPS (128-bit variable version)
DEST[31:0] := Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] := Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] := Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] := Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPERMILPS __m512 _mm512_permute_ps( __m512 a, int imm);
VPERMILPS __m512 __m512_permute_ps( __m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 __m512_mask_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 __m256_permute_ps( __m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __m256_mask_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 __m128_permute_ps( __m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 __m128_mask_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m512 __m512_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 __m512_mask_permutevar_ps( __m512 i, __m512i i, __m512 a);
VPERMILPS __m512 __m512_permutevar_ps( __m512 i, __m512i i, __m512 a);
VPERMILPS __m256 __m256_permutevar_ps( __m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 __m256_mask_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 __m128_permutevar_ps( __m128 s, __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 __m128_mask_permutevar_ps( __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 __m128_permute_ps( __m128 s, __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 __m128_permutevar_ps( __m128 i, __m128i i, __m128 control);
VPERMILPS __m128i __m128_permutevar_ps( __m128 a, __m128i control);
VPERMILPS __m256i __m256_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 != 111B and with imm8.
VPERMPD—Permute Double-Precision Floating-Point Elements

### 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 qword element in the source operand.

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

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

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

**VPERMPD (EVEX - imm8 control forms)**

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

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

\[ i := j \times 64 \]

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

\[ \text{THEN } \text{TMP\_SRC}[i+63:i] := \text{SRC}[63:0]; \]

ELSE \( \text{TMP\_SRC}[i+63:i] := \text{SRC}[i+63:i]; \)

FI;

ENDFOR;

\[ \text{TMP\_DEST}[63:0] := (\text{TMP\_SRC}[256:0] >> (\text{IMM8}[1:0] \times 64))[63:0]; \]

\[ \text{TMP\_DEST}[127:64] := (\text{TMP\_SRC}[256:0] >> (\text{IMM8}[3:2] \times 64))[63:0]; \]

\[ \text{TMP\_DEST}[191:128] := (\text{TMP\_SRC}[256:0] >> (\text{IMM8}[5:4] \times 64))[63:0]; \]

\[ \text{TMP\_DEST}[255:192] := (\text{TMP\_SRC}[256:0] >> (\text{IMM8}[7:6] \times 64))[63:0]; \]

IF \( \text{VL} \geq 512 \)

\[ \text{TMP\_DEST}[319:256] := (\text{TMP\_SRC}[511:256] >> (\text{IMM8}[1:0] \times 64))[63:0]; \]

\[ \text{TMP\_DEST}[383:320] := (\text{TMP\_SRC}[511:256] >> (\text{IMM8}[3:2] \times 64))[63:0]; \]

\[ \text{TMP\_DEST}[447:384] := (\text{TMP\_SRC}[511:256] >> (\text{IMM8}[5:4] \times 64))[63:0]; \]

\[ \text{TMP\_DEST}[511:448] := (\text{TMP\_SRC}[511:256] >> (\text{IMM8}[7:6] \times 64))[63:0]; \]

FI;

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

\[ i := j \times 64 \]

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

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

ELSE

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

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

ELSE

\[ \text{DEST}[i+63:i] := 0 \text{ ;zeroing-masking} \]

FI;

FI;

ENDFOR

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

**VPERMPD (EVEX - vector control forms)**

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

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

\[ i := j \times 64 \]

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

\[ \text{THEN } \text{TMP\_SRC2}[i+63:i] := \text{SRC2}[63:0]; \]

ELSE \( \text{TMP\_SRC2}[i+63:i] := \text{SRC2}[i+63:i]; \)

FI;

ENDFOR;

IF \( \text{VL} = 256 \)

\[ \text{TMP\_DEST}[63:0] := (\text{TMP\_SRC2}[255:0] >> (\text{SRC1}[1:0] \times 64))[63:0]; \]

\[ \text{TMP\_DEST}[127:64] := (\text{TMP\_SRC2}[255:0] >> (\text{SRC1}[65:64] \times 64))[63:0]; \]

\[ \text{TMP\_DEST}[191:128] := (\text{TMP\_SRC2}[255:0] >> (\text{SRC1}[129:128] \times 64))[63:0]; \]

\[ \text{TMP\_DEST}[255:192] := (\text{TMP\_SRC2}[255:0] >> (\text{SRC1}[193:192] \times 64))[63:0]; \]

FI;

IF \( \text{VL} = 512 \)

\[ \text{TMP\_DEST}[63:0] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[2:0] \times 64))[63:0]; \]
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* ; merging-masking
         THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+63:i] := 0 ;zeroing-masking
      FI;
   FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMPD (VEX.256 encoded version)
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPERMPD __m512d _mm512_permutex_pd( __m512d a, int imm);
VPERMPD __m512d _mm512_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_maskz_permutex_pd( __mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_permutexvar_pd(__m512i i, __m512d a);
VPERMPD __m512d _mm512_mask_permutexvar_pd(__m512d s, __mmask16 k, __m512i i, __m512d a);
VPERMPD __m512d _mm512_maskz_permutexvar_pd( __mmask16 k, __m512i i, __m512d a);
VPERMPD __m256d _mm256_permutex_epi64( __m256d a, int imm);
VPERMPD __m256d _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_maskz_permutex_epi64( __mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_permutexvar_epi64(__m256i i, __m256d a);
VPERMPD __m256d _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d _mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i i, __m256d a);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see 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 EVEX.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</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 indexes in zmm2 and store the result in zmm1 subject to write mask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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>
</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

\[
\begin{align*}
\text{IF } VL = 512 & \\
\text{IF } VL = 512 & \\
\text{TMP\_DEST}[31:0] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[3:0] * 32))[31:0]; \\
\text{TMP\_DEST}[63:32] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[35:32] * 32))[31:0]; \\
\text{TMP\_DEST}[95:64] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[67:64] * 32))[31:0]; \\
\text{TMP\_DEST}[127:96] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[99:96] * 32))[31:0]; \\
\text{TMP\_DEST}[159:128] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[131:128] * 32))[31:0]; \\
\text{TMP\_DEST}[191:160] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[163:160] * 32))[31:0]; \\
\text{TMP\_DEST}[223:192] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[195:192] * 32))[31:0]; \\
\text{TMP\_DEST}[255:224] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[227:224] * 32))[31:0]; \\
\text{TMP\_DEST}[287:256] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[259:256] * 32))[31:0]; \\
\text{TMP\_DEST}[319:288] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[291:288] * 32))[31:0]; \\
\text{TMP\_DEST}[351:320] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[323:320] * 32))[31:0]; \\
\text{TMP\_DEST}[383:352] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[355:352] * 32))[31:0]; \\
\text{TMP\_DEST}[415:384] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[387:384] * 32))[31:0]; \\
\text{TMP\_DEST}[447:416] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[419:416] * 32))[31:0]; \\
\text{TMP\_DEST}[479:448] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[451:448] * 32))[31:0]; \\
\text{TMP\_DEST}[511:480] := (\text{TMP\_SRC2}[511:0] >> (\text{SRC1}[483:480] * 32))[31:0]; \\
\end{align*}
\]

VPERMPS (VEX.256 encoded version)

\[
\begin{align*}
\text{DEST}[31:0] := (\text{SRC2}[255:0] >> (\text{SRC1}[2:0] * 32))[31:0]; \\
\text{DEST}[63:32] := (\text{SRC2}[255:0] >> (\text{SRC1}[34:32] * 32))[31:0]; \\
\text{DEST}[95:64] := (\text{SRC2}[255:0] >> (\text{SRC1}[66:64] * 32))[31:0]; \\
\text{DEST}[127:96] := (\text{SRC2}[255:0] >> (\text{SRC1}[98:96] * 32))[31:0]; \\
\text{DEST}[159:128] := (\text{SRC2}[255:0] >> (\text{SRC1}[130:128] * 32))[31:0]; \\
\text{DEST}[191:160] := (\text{SRC2}[255:0] >> (\text{SRC1}[162:160] * 32))[31:0]; \\
\text{DEST}[223:192] := (\text{SRC2}[255:0] >> (\text{SRC1}[194:192] * 32))[31:0]; \\
\text{DEST}[255:224] := (\text{SRC2}[255:0] >> (\text{SRC1}[226:224] * 32))[31:0]; \\
\text{DEST}[MAXVL-1:256] := 0
\end{align*}
\]
Intel C/C++ Compiler Intrinsic Equivalent
VPERMPS __m512 __mm512_permutexvar_ps(__m512i, __m512 a);
VPERMPS __m512 __mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i, __m512 a);
VPERMPS __m512 __mm512_maskz_permutexvar_ps( __mmask16 k, __m512i, __m512 a);
VPERMPS __m256 __mm256_permutexvar_ps(__m256i, __m256 a);
VPERMPS __m256 __mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256i, __m256 a);
VPERMPS __m256 __mm256_maskz_permutexvar_ps( __mmask8 k, __m256i, __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

**Opcode/Instruction**

<table>
<thead>
<tr>
<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>
</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>
</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>
</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>
</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>
</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>Imm8</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</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**

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.
INSTRUCTION SET REFERENCE, V-Z

Operation

VPERMQ (EVEX - imm8 control forms)

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

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

  TMP_DEST[63:0] := (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
  TMP_DEST[127:64] := (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];

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[i+63:i];
    ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
  FI;
ENDFOR;

IF VL = 256
  TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
  TMP_DEST[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];
```c
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* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0 ;zeroing-masking
    FI;
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMQ (VEX.256 encoded version)
DEST[63:0] := (SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] := (SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[255:192] := (SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPERMQ __m512i _mm512_permutex_epi64( __m512i a, int imm);
VPERMQ __m512i _mm512_mask_permutex_epi64(__m512s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_maskz_permutex_epi64(__mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_permutexvar_epi64( __m512i a, __m512i b);
VPERMQ __m512i _mm512_mask_permutexvar_epi64(__m512s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i _mm512_maskz_permutexvar_epi64(__mmask8 k, __m512i a, __m512i b);
VPERMQ __m256i _mm256_permutex_epi64( __m256i a, int imm);
VPERMQ __m256i _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_maskz_permutex_epi64(__mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_permutexvar_epi64( __m256i a, __m256i b);
VPERMQ __m256i _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPERMQ __m256i _mm256_maskz_permutexvar_epi64( __mmask8 k, __m256i a, __m256i b);

SIMD Floating-Point Exceptions
None

Other Exceptions
Non-EVEX-encoded instruction, see 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

### 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

Permutates 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 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 7D /r VPERMT2W 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 ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w1 7D /r VPERMT2W 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 zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w0 7E /r VPERMT2D 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 xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w0 7E /r VPERMT2D 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 ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 7E /r VPERMT2D 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 zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w1 7E /r VPERMT2Q 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 xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 7E /r VPERMT2Q 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 ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w1 7E /r VPERMT2Q 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 zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w0 7F /r VPERMT2PS 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 xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w0 7F /r VPERMT2PS 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 ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w0 7F /r VPERMT2PS 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 zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.w1 7F /r VPERMT2PD xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.w1 7F /r VPERMT2PD ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td>B 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.w1 7F /r VPERMT2PD zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</td>
<td>B 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>
<td></td>
</tr>
</tbody>
</table>
**Description**

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

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

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands. If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0]/[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
FL;

IF VL = 256
   id := 3
FL;

IF VL = 512
   id := 4
FL;

TMP_DEST := DEST
FOR j := 0 TO KL-1
   i := j * 16
   off := 16 * SRC1[i+id]
   IF k1[j] OR *no writemask*
      THEN
         DEST[i+15:j]=SRC1[i+id+1] ? SRC2[off+15:off]
      ELSE
         IF *merging-masking* ; merging-masking
            THEN *DEST[i+15:j] remains unchanged*
         ELSE ; zeroing-masking

---

**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 (r,w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg/r (r)</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (r, w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg/r (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>
DEST[i+15:i] := 0
FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPERMT2D/VPERMT2PS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
id := 1
FI;
IF VL = 256
id := 2
FI;
IF VL = 512
id := 3
FI;
TMP_DEST := DEST
FOR j := 0 TO KL-1
   i := j * 32
   off := 32*SRC1[i+id]
   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:i]

IF k1[] OR *no writemask*
THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN
    ELSE
    FI
ELSE
    IF *merging-masking* ; merging-masking
    THEN *DEST[i+63: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 __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512d idx, __m512d b);
VPERMT2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512d idx, __m512d b);
VPERMT2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512d idx, __mmask8 k, __m512d b);
VPERMT2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512d idx, __m512d b);
VPERMT2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256d idx, __m256d b);
VPERMT2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256d idx, __m256d b);
VPERMT2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256d idx, __mmask8 k, __m256d b);
VPERMT2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256d idx, __m256d b);
VPERMT2PD __m128d _mm_permutex2var_pd(__m128d a, __m128d idx, __m128d b);
VPERMT2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128d idx, __m128d b);
VPERMT2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128d idx, __mmask8 k, __m128d b);
VPERMT2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128d idx, __m128d b);
VPERMT2PS __m512 __mm512_permutex2var_ps(__m512 a, __m512d idx, __m512 b);
VPERMT2PS __m512 __mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512d idx, __m512 b);
VPERMT2PS __m512 __mm512_mask2_permutex2var_ps(__m512 a, __m512d idx, __mmask16 k, __m512 b);
VPERMT2PS __m512 __mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512d idx, __m512 b);
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_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMT2PS __m128 _mm128_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm128_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm128_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);

VPERMT2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMT2Q __m128i _mm128i_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm128i_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm128i_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i 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".
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 256 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*:

\[\text{DEST.byte}[j] := \text{SRC.byte}[k];\]

\[k := k + 1\]

ELSE:

IF *merging-masking*:

*DEST.byte*[j] remains unchanged*

ELSE:

DEST.byte[j] := 0

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

**VPEXPANDW**

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

\[k := 0\]

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

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

\[\text{DEST.word}[j] := \text{SRC.word}[k];\]

\[k := k + 1\]

ELSE:

IF *merging-masking*:

*DEST.word*[j] remains unchanged*

ELSE:

DEST.word[j] := 0

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

VPEXPAND __m128i _mm_mask_expand_epi8(__m128i, __mmask16, __m128i);
VPEXPAND __m256i __m256_maskz_expand_epi8(__mmask16, __m128i);
VPEXPAND __m128i __m_mm_maskzexpand_epi8(__m128i, __mmask16, const void*);
VPEXPAND __m128i __m_mm_maskzexpandloadu_epi8(__m128i, __mmask16, const void*);
VPEXPAND __m256i __m256_maskzexpand_epi8(__m256i, __mmask32, __m256i);
VPEXPAND __m256i __m256_maskzexpandloadu_epi8(__m256i, __mmask32, const void*);
VPEXPAND __m256i __m256_maskzexpand_epi8(__m256i, __mmask32, __m256i);
VPEXPAND __m256i __m256_maskzexpandloadu_epi8(__m256i, __mmask32, const void*);
VPEXPAND __m128i __m_mm_maskzexpand_epi16(__m128i, __mmask8, __m128i);
VPEXPAND __m256i __m256_maskzexpand_epi16(__m256i, __mmask16, __m256i);
VPEXPAND __m256i __m256_maskzexpandloadu_epi16(__m256i, __mmask16, const void*);
VPEXPAND __m512i __m512_maskzexpand_epi16(__m512i, __mmask32, __m512i);
VPEXPAND __m512i __m512_maskzexpandloadu_epi16(__m512i, __mmask32, const void*);
VPEXPANDW __m128i __m_mm_maskzexpand_epi16(__m128i, __mmask8, __m128i);
VPEXPANDW __m256i __m256_maskzexpandloadu_epi16(__m128i, __mmask8, const void*);
VPEXPANDW __m256i __m256_maskzexpand_epi16(__m256i, __mmask8, __m256i);
VPEXPANDW __m256i __m256_maskzexpandloadu_epi16(__m256i, __mmask8, const void*);
VPEXPANDW __m512i __m512_maskzexpand_epi16(__m512i, __mmask32, __m512i);
VPEXPANDW __m512i __m512_maskzexpandloadu_epi16(__m512i, __mmask32, const void*);
VPEXPANDW __m512i __m512_maskzexpand_epi16(__m512i, __mmask32, __m512i);
VPEXPANDW __m512i __m512_maskzexpandloadu_epi16(__m512i, __mmask32, const void*);

SIMD Floating-Point Exceptions

None.

Other Exceptions

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

VPEXPANDB/VPEXPANDW — Expand Byte/Word Values
VPEXPANDD—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode Support</th>
<th>CPuid Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 89 / r VPEXPANDD xmm1 {k1}[z], xmm2/m128</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 89 / r VPEXPANDD ymm1 {k1}[z], ymm2/m256</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 89 / r VPEXPANDD zmm1 {k1}[z], zmm2/m512</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.</td>
</tr>
</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)

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

\[k := 0\]

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

\[i := j \times 32\]

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

\[\text{THEN}\]

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

\[k := k + 32\]

\[\text{ELSE}\]

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

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

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

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

\[\text{FI}\]

\[\text{FI}\]

\[\text{ENDFOR}\]

\[\text{DEST}[\text{MAXVL}-1:VL] := 0\]
**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_maskz_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_maskz_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i _mm256_maskz_expand_epi32(__mmask8 k, __m256i a);
VPEXPANDD __m128i _mm_maskz_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i _mm_maskz_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

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128:66.0F38.W1 89 /r VPEXPANDQ xmm1 {k1}[z], xmm2/m128</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256:66.0F38.W1 89 /r VPEXPANDQ ymm1 {k1}[z], ymm2/m256</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512:66.0F38.W1 89 /r VPEXPANDQ zmm1 {k1}[z], zmm2/m512</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Expand packed quad-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 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

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

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

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

Operation

VPEXPANDQ (EVEX encoded versions)

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

VPEXPANDQ __m512i __m512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VPEXPANDQ __m512i __m512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m512i __m512_mask_expand_epi64(__m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m512i __m512_maskz_expand_epi64(__mmask8 k, __m512i a);
VPEXPANDQ __m256i __m256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i __m256_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m256i __m256_maskz_expand_epi64(__mmask8 k, __m256i a);
VPEXPANDQ __m256i __m256_maskz_expand_epi64(__mmask8 k, __m256i a);
VPEXPANDQ __m128i __m128_maskz_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VPEXPANDQ __m128i __m128_maskz_expandloadu_epi64(__mmask8 k, __m128i a);
VPEXPANDQ __m128i __m128_maskz_expand_epi64(__mmask8 k, __m128i a);
VPEXPANDQ __m128i __m128_maskz_expand_epi64(__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.
INSTRUCTION SET REFERENCE, V-Z

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>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMV</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:Index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero’ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zero the upper 64 bits of the destination.
VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

INSTRUCTION SET REFERENCE, V-Z

Vol. 2C 5-405

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;
VPGATHERDD (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 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-1:64] := 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;
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, __m128i 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.
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]], 1)
        k1[i] := 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[i] := 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**

- \texttt{VPGATHERDD \_m512i \_mm512\_i32gather\_epi32(\_m512i vdx, void * base, int scale);}
- \texttt{VPGATHERDD \_m512i \_mm512\_mask\_i32gather\_epi32(\_m512i s, \_mmask16 k, \_m512i vdx, void * base, int scale);}
- \texttt{VPGATHERDD \_m256i \_mm256\_mmask\_i32gather\_epi32(\_m256i s, \_mmask8 k, \_m256i vdx, void * base, int scale);}
- \texttt{VPGATHERDD \_m128i \_mm\_mmask\_i32gather\_epi32(\_m128i s, \_mmask8 k, \_m128i vdx, void * base, int scale);}
- \texttt{VPGATHERDQ \_m512i \_mm512\_i32logather\_epi64(\_m512i vdx, void * base, int scale);}
- \texttt{VPGATHERDQ \_m512i \_mm512\_mask\_i32logather\_epi64(\_m512i s, \_mmask8 k, \_m512i vdx, void * base, int scale);}
- \texttt{VPGATHERDQ \_m256i \_mm256\_mmask\_i32logather\_epi64(\_m256i s, \_mmask8 k, \_m256i vdx, void * base, int scale);}
- \texttt{VPGATHERDQ \_m128i \_mm\_mmask\_i32logather\_epi64(\_m128i 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”. 
VPGATHERDQ/VPGATHERQQ — Gather Packed Qword 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.W1 90 /r VPGATHERDQ xmm1, vm32x, xmm2</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 VPGATHERQQ xmm1, vm64x, xmm2</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 VPGATHERDQ ymm1, vm32x, ymm2</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 VPGATHERQQ ymm1, vm64y, ymm2</td>
<td>A</td>
<td>V/V</td>
<td>AVX2</td>
<td>Using qword indices specified in vm64y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ModRM:reg (r,w)</td>
<td>BaseReg (R): VSIB:base, VectorReg(R): VSIB:index</td>
<td>VEX.vvvv (r, w)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element’s mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.
VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices

INSTRUCTION SET REFERENCE, V-Z

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:
- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

DEST := SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB[7:6];
DISP: optional 1, 4 byte displacement;
MASK := SRC3;

VPGATHERDQ (VEX.128 version)
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;
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[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/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 VPGATHERQD xmm1 [k1], vm64x</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 91 /vsib VPGATHERQD xmm1 [k1], vm64y</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 91 /vsib VPGATHERQD ymm1 [k1], vm64z</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 91 /vsib VPGATHERQQ xmm1 [k1], vm64x</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 91 /vsib VPGATHERQQ ymm1 [k1], vm64y</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 91 /vsib VPGATHERQQ zmm1 [k1], vm64z</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.</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[BASE_ADDR + (VINDEX[k+63:k] * SCALE + DISP)], 1)
k1[j] := 0
ELSE *DEST[i+31:i] := remains unchanged* ; Only merging masking is allowed
Fi;
ENDFOR
k1[MAX_KL-1:KL] := 0
DEST[MAXVL-1:VL/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[BASE_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 __m128i _mm_mask_i64gather_epi64(__m128i 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".

VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices
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</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>VPLZCNTD xmm1 {k1}{z}, xmm2/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 44 /r</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>VPLZCNTD ymm1 {k1}{z}, ymm2/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 44 /r</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>VPLZCNTD zmm1 {k1}{z}, zmm2/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 44 /r</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>VPLZCNTQ xmm1 {k1}{z}, xmm2/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 44 /r</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>VPLZCNTQ ymm1 {k1}{z}, ymm2/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 44 /r</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>
<tr>
<td>VPLZCNTQ zmm1 {k1}{z}, zmm2/m512/m64bcst</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**

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 Support</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</td>
<td>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>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 B5 /r VPMADD52HUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</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 high 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 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst</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 high 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.vvv (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];
FI;
Temp128[127:0] := ZeroExtend64(src1[i+51:i]) * tsrc2[63:0];
Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[103:52]);
DEST[i+63:i] := Temp2[63:0];
ELSE

IF *zeroing-masking* THEN

DEST[i+63:i] := 0;
ELSE *merge-masking*

DEST[i+63:i] is unchanged;
FI;
FI;
ENDFOR

DEST[MAX_VL-1:VL] := 0

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

VPMADD52HUQ __m512i _mm512_madd52hi_epu64( __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_mask_madd52hi_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m512i _mm512_maskz_madd52hi_epu64( __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52HUQ __m256i _mm256_madd52hi_epu64( __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_mask_madd52hi_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m256i _mm256_maskz_madd52hi_epu64( __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52HUQ __m128i _mm128i_madd52hi_epu64( __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm128i_mask_madd52hi_epu64( __m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52HUQ __m128i _mm128i_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 VPMADD52LUQ xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst</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 VPMADD52LUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst</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 VPMADD52LUQ zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst</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 * 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+64]);
        FI;
        Temp128[127:0] := ZeroExtend64(src1[i+51:i+64]) * tsrc2[63:0];
        Temp2[63:0] := DEST[i+63:i+128] + ZeroExtend64(temp128[51:0]);
        DEST[i+63:i+128] := Temp2[63:0];
    ELSE
        IF *zeroing-masking* THEN
            DEST[i+63:i+128] := 0;
        ELSE *merge-masking*
            DEST[i+63:i+128] is unchanged;
        FI;
    FI;
ENDFOR
DEST[MAX_VL-1:VL] := 0;

Intel C/C++ Compiler Intrinsic Equivalent

VPMADD52LUQ __m512i _mm512_madd52lo_epu64( __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_mask_madd52lo_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m512i _mm512_maskz_madd52lo_epu64( __mmask8 k, __m512i a, __m512i b, __m512i c);
VPMADD52LUQ __m256i _mm256_madd52lo_epu64( __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_mask_madd52lo_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m256i _mm256_maskz_madd52lo_epu64( __mmask8 k, __m256i a, __m256i b, __m256i c);
VPMADD52LUQ __m128i _mm128_madd52lo_epu64( __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm128_mask_madd52lo_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b, __m128i c);
VPMADD52LUQ __m128i _mm128_maskz_madd52lo_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”.
VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32-bit Mode</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 8C /r VPMASKMOVD xmm1, xmm2, m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load dword values from (m_{128}) using mask in (xmm2) and store in (xmm1).</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 8C /r VPMASKMOVD ymm1, ymm2, m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load dword values from (m_{256}) using mask in (ymm2) and store in (ymm1).</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 8C /r VPMASKMOVQ xmm1, xmm2, m128</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load qword values from (m_{128}) using mask in (xmm2) and store in (xmm1).</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 8C /r VPMASKMOVQ ymm1, ymm2, m256</td>
<td>RVM</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally load qword values from (m_{256}) using mask in (ymm2) and store in (ymm1).</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 8E /r VPMASKMOVD m128, xmm1, xmm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally store dword values from (xmm2) using mask in (xmm1).</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 8E /r VPMASKMOVD m256, ymm1, ymm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally store dword values from (ymm2) using mask in (ymm1).</td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 8E /r VPMASKMOVQ m128, xmm1, xmm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally store qword values from (xmm2) using mask in (xmm1).</td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 8E /r VPMASKMOVQ m256, ymm1, ymm2</td>
<td>MVR</td>
<td>V/V</td>
<td>AVX2</td>
<td>Conditionally store qword values from (ymm2) using mask in (ymm1).</td>
</tr>
</tbody>
</table>

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 imple-
mentation 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**

\[\text{DEST}[31:0] := \text{IF (SRC1[31]) Load}_32(\text{mem}) \text{ ELSE 0}\]
\[\text{DEST}[63:32] := \text{IF (SRC1}[63]) \text{ Load}_32(\text{mem} + 4) \text{ ELSE 0}\]
\[\text{DEST}[95:64] := \text{IF (SRC1}[95]) \text{ Load}_32(\text{mem} + 8) \text{ ELSE 0}\]
\[\text{DEST}[127:96] := \text{IF (SRC1}[127]) \text{ Load}_32(\text{mem} + 12) \text{ ELSE 0}\]
\[\text{DEST}[159:128] := \text{IF (SRC1}[159]) \text{ Load}_32(\text{mem} + 16) \text{ ELSE 0}\]
\[\text{DEST}[191:160] := \text{IF (SRC1}[191]) \text{ Load}_32(\text{mem} + 20) \text{ ELSE 0}\]
\[\text{DEST}[223:192] := \text{IF (SRC1}[223]) \text{ Load}_32(\text{mem} + 24) \text{ ELSE 0}\]
\[\text{DEST}[255:224] := \text{IF (SRC1}[255]) \text{ Load}_32(\text{mem} + 28) \text{ ELSE 0}\]

**VPMASKMOVD - 128-bit load**

\[\text{DEST}[31:0] := \text{IF (SRC1}[31]) \text{ Load}_32(\text{mem}) \text{ ELSE 0}\]
\[\text{DEST}[63:32] := \text{IF (SRC1}[63]) \text{ Load}_32(\text{mem} + 4) \text{ ELSE 0}\]
\[\text{DEST}[95:64] := \text{IF (SRC1}[95]) \text{ Load}_32(\text{mem} + 8) \text{ ELSE 0}\]
\[\text{DEST}[127:97] := \text{IF (SRC1}[127]) \text{ Load}_32(\text{mem} + 12) \text{ ELSE 0}\]
\[\text{DEST}[\text{MAXVL}-1:128] := 0\]

**VPMASKMOVDQ - 256-bit load**

\[\text{DEST}[63:0] := \text{IF (SRC1}[63]) \text{ Load}_64(\text{mem}) \text{ ELSE 0}\]
\[\text{DEST}[127:64] := \text{IF (SRC1}[127]) \text{ Load}_64(\text{mem} + 8) \text{ ELSE 0}\]
\[\text{DEST}[195:128] := \text{IF (SRC1}[191]) \text{ Load}_64(\text{mem} + 16) \text{ ELSE 0}\]
\[\text{DEST}[255:196] := \text{IF (SRC1}[255]) \text{ Load}_64(\text{mem} + 24) \text{ ELSE 0}\]

**VPMASKMOVDQ - 128-bit load**

\[\text{DEST}[63:0] := \text{IF (SRC1}[63]) \text{ Load}_64(\text{mem}) \text{ ELSE 0}\]
\[\text{DEST}[127:64] := \text{IF (SRC1}[127]) \text{ Load}_64(\text{mem} + 16) \text{ ELSE 0}\]
\[\text{DEST}[\text{MAXVL}-1:128] := 0\]

**VPMASKMOVD - 256-bit store**

\[\text{IF (SRC1}[31]) \text{ DEST}[31:0] := \text{SRC2}[31:0]\]
\[\text{IF (SRC1}[63]) \text{ DEST}[63:32] := \text{SRC2}[63:32]\]
\[\text{IF (SRC1}[95]) \text{ DEST}[95:64] := \text{SRC2}[95:64]\]
\[\text{IF (SRC1}[127]) \text{ DEST}[127:96] := \text{SRC2}[127:96]\]
\[\text{IF (SRC1}[159]) \text{ DEST}[159:128] := \text{SRC2}[159:128]\]
\[\text{IF (SRC1}[191]) \text{ DEST}[191:160] := \text{SRC2}[191:160]\]
\[\text{IF (SRC1}[223]) \text{ DEST}[223:192] := \text{SRC2}[223:192]\]
\[\text{IF (SRC1}[255]) \text{ DEST}[255:224] := \text{SRC2}[255:224]\]
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]

VPMASKMOVD - 128-bit store
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent
VPMASKMOVD: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)
VPMASKMOVD: void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)
VPMASKMOVD: __m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);
VPMASKMOVD: void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m128i b);
VPMASKMOVD: __m128i _mm_maskload_epi32(int const *a, __m128i mask)
VPMASKMOVD: void _mm_maskstore_epi32(int *a, __m128i mask, __m128i b)
VPMASKMOVD: __m128i _mm_maskload_epi64(__int const *a, __m128i mask);
VPMASKMOVD: void _mm_maskstore_epi64(__int64 *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

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/ En</th>
<th>64/32 bit Mode</th>
<th>CPUID Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 29 /r VPMOVB2M k1, xmm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 29 /r VPMOVB2M k1, ymm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 29 /r VPMOVB2M k1, zmm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 29 /r VPMOVw2M k1, xmm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 29 /r VPMOVw2M k1, ymm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 29 /r VPMOVw2M k1, zmm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 39 /r VPMOVD2M k1, xmm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 39 /r VPMOVD2M k1, ymm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 39 /r VPMOVD2M k1, zmm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 39 /r VPMOVQ2M k1, xmm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 39 /r VPMOVQ2M k1, ymm1</td>
<td>RM 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 39 /r VPMOVQ2M k1, zmm1</td>
<td>RM 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>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

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

### Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register. EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.
Operation

VPMOVB2M (EVEX encoded versions)

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

FOR \(j := 0 \) TO \(KL-1\)
    \(i := j * 8\)
    IF \(\text{SRC}[i+7]\)
        THEN \(\text{DEST}[j] := 1\)
        ELSE \(\text{DEST}[j] := 0\)
    FI;
ENDFOR

\(\text{DEST}[\text{MAX}_K L-1:KL] := 0\)

VPMOVW2M (EVEX encoded versions)

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

FOR \(j := 0 \) TO \(KL-1\)
    \(i := j * 16\)
    IF \(\text{SRC}[i+15]\)
        THEN \(\text{DEST}[j] := 1\)
        ELSE \(\text{DEST}[j] := 0\)
    FI;
ENDFOR

\(\text{DEST}[\text{MAX}_K L-1:KL] := 0\)

VPMOVD2M (EVEX encoded versions)

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

FOR \(j := 0 \) TO \(KL-1\)
    \(i := j * 32\)
    IF \(\text{SRC}[i+31]\)
        THEN \(\text{DEST}[j] := 1\)
        ELSE \(\text{DEST}[j] := 0\)
    FI;
ENDFOR

\(\text{DEST}[\text{MAX}_K L-1:KL] := 0\)

VPMOVQ2M (EVEX encoded versions)

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

FOR \(j := 0 \) TO \(KL-1\)
    \(i := j * 64\)
    IF \(\text{SRC}[i+63]\)
        THEN \(\text{DEST}[j] := 1\)
        ELSE \(\text{DEST}[j] := 0\)
    FI;
ENDFOR

\(\text{DEST}[\text{MAX}_K L-1:KL] := 0\)
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</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</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</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</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</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</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</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed double-word integers from zmm2 into 16 packed byte integers in xmm1/m128 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 21 /r</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed signed double-word integers from zmm2 into 16 packed signed byte integers in xmm1/m128 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 11 /r</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned byte integers in xmm1/m128 using 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>ModRMr/m (w)</td>
<td>ModRMreg (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.vvv 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 * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN Dest[i+7:i] := TruncateDoubleWordToByte (Src[m+31:m])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *Dest[i+7:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                Dest[i+7:i] := 0
            FI
    FI;
ENDFOR

DEST[MAXVL-1:VL/4] := 0;

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 * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN Dest[i+7:i] := TruncateDoubleWordToByte (Src[m+31:m])
        ELSE *Dest[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

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 * 8
    m := j * 32
    IF k1[j] OR *no writemask*
        THEN Dest[i+7:i] := SaturateSignedDoubleWordToByte (Src[m+31:m])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *Dest[i+7:i] remains unchanged*
            ELSE *zeroing-masking* ; zeroing-masking
                Dest[i+7:i] := 0
            FI
    FI;
ENDFOR

DEST[MAXVL-1:VL/4] := 0;
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 * 8
  m := j * 32
  IF k1[j] OR "no writemask"
    THEN DEST[i+7:i] := SaturateSignedDoubleWordToByte (SRC[m+31:m])
    ELSE *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

VPMOVUSDB instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 32
  IF k1[j] OR "no writemask"
    THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] := 0
      FI
  FI
ENDFOR

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_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVSD __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVSD __m128i _mm512_mask_cvtepi32_epi8(__m512i s, __mmask16 k, __m512i a);
VPMOVSD __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVSD __m128i _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVSD __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVSD __m128i _mm512_mask_cvtsepi32_epi8(__m512i s, __mmask16 k, __m512i a);
VPMOVSD __m128i _mm512_maskz_cvtsepi32_epi8(__mmask16 k, __m512i a);
VPMOVSD __m128i _mm512_mask_cvtusepi32_epi8(__m512i s, __mmask16 k, __m512i a);
VPMOVSD __m128i _mm512_maskz_cvtusepi32_epi8(__mmask16 k, __m512i a);
VPMOVSD __m128i _mm256_cvtepi32_epi8(__m256i a);
VPMOVSD __m128i _mm256_mask_cvtepi32_epi8(__m256i a, __mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_mask_cvtepi32_storeu_epi8(void * d, __mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_mask_cvtsepi32_epi8(__m256i a, __mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_maskz_cvtsepi32_epi8(__mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_mask_cvtusepi32_epi8(__m256i a, __mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_maskz_cvtusepi32_epi8(__mmask8 k, __m256i b);
VPMOVSD __m128i _mm_cvtusepi32_epi8(__m128i a);
VPMOVSD __m128i _mm_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSD __m128i _mm_maskz_cvtusepi32_epi8(__mmask8 k, __m128i b);
VPMOVSD __m128i _mm_mask_cvtusepi32_storeu_epi8(void * d, __mmask8 k, __m128i b);
VPMOVSD __m128i _mm256_cvtusepi32_epi8(__m256i a);
VPMOVSD __m128i _mm256_mask_cvtusepi32_epi8(__m256i a, __mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_maskz_cvtusepi32_epi8(__mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_mask_cvtusepi32_storeu_epi8(void * d, __mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_maskz_cvtusepi32_epi8(__mmask8 k, __m256i b);
VPMOVSD __m128i _mm_cvtepi32_epi8(__m128i a);
VPMOVSD __m128i _mm_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSD __m128i _mm_maskz_cvtepi32_epi8(__mmask8 k, __m128i b);
VPMOVSD __m128i _mm_mask_cvtepi32_storeu_epi8(void * d, __mmask8 k, __m128i b);
VPMOVSD __m128i _mm256_cvtepi32_epi8(__m256i a);
VPMOVSD __m128i _mm256_mask_cvtepi32_epi8(__m256i a, __mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i b);
VPMOVSD __m128i _mm256_mask_cvtepi32_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.
### 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 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 23 /r VPMOVSDw xmm1/m64 [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 word integers in xmm1/m64 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 13 /r VPMOVUSDw xmm1/m64 [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 word integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 33 /r VPMOVDw xmm1/m128 [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 word integers in xmm1/m128 with truncation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 23 /r VPMOVSDw xmm1/m128 [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 word integers in xmm1/m128 using signed saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 13 /r VPMOVUSDw xmm1/m128 [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 word integers in xmm1/m128 using unsigned saturation under writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 33 /r VPMOVDw ymm1/m256 [k1][z], zmm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 23 /r VPMOVSDw ymm1/m256 [k1][z], zmm2</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 13 /r VPMOVUSDw ymm1/m256 [k1][z], zmm2</td>
<td>A 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>
<td></td>
</tr>
</tbody>
</table>

### 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>

### Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

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

VPMOVDw instruction (EVEX encoded versions) when dest is a register

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking*; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking*; zeroing-masking
        DEST[i+15:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;

VPMOVDw instruction (EVEX encoded versions) when dest is memory

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      *DEST[i+15:i] remains unchanged*; merging-masking
    FI;
ENDFOR

VPMOVSDw instruction (EVEX encoded versions) when dest is a register

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateSignedDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking*; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE *zeroing-masking*; zeroing-masking
        DEST[i+15:i] := 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;
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_maskword_cvtepi32_epi16( __m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i __mm512_storew_epi16( void * d, __mmask16 k, __m512i a);
VPMOVDW __m256i __mm512_cvtsepi32_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 __m256i __mm512_maskzword_cvtepi32_epi16( __m256i s, __mmask16 k, __m512i a);
VPMOVDW __m512i __mm512_cvtsepi32_epi16( __m512i a);
VPMOVDW __m512i __mm512_mask_cvtepi32_epi16( __m512i s, __mmask16 k, __m512i a);
VPMOVDW __m512i __mm512_maskz_cvtepi32_epi16( __mmask16 k, __m512i a);
VPMOVDW __m512i __mm512_maskzword_cvtepi32_epi16( __m512i s, __mmask16 k, __m512i a);
VPMOVDW __m128i __mm512_cvtsepi32_epi16( __m128i a);
VPMOVDW __m128i __mm512_mask_cvtepi32_epi16( __m128i s, __mmask8 k, __m128i a);
VPMOVDW __m128i __mm512_maskz_cvtepi32_epi16( __mmask8 k, __m128i a);
VPMOVDW __m128i __mm512_maskzword_cvtepi32_epi16( __m128i s, __mmask8 k, __m128i a);
VPMOVDW __m256i __mm256_cvtepi32_epi16( __m256i a);
VPMOVDW __m256i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask16 k, __m256i a);
VPMOVDW __m256i __mm256_maskword_cvtepi32_epi16( __m256i s, __mmask16 k, __m256i a);
VPMOVDW __m256i __mm256_storew_epi16( void * d, __mmask16 k, __m256i a);
VPMOVDW __m256i __mm256_cvtsepi32_epi16( __m256i a);
VPMOVDW __m256i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask16 k, __m256i a);
VPMOVDW __m256i __mm256_maskz_cvtepi32_epi16( __mmask16 k, __m256i a);
VPMOVDW __m256i __mm256_maskzword_cvtepi32_epi16( __m256i s, __mmask16 k, __m256i a);
VPMOVDW __m128i __mm256_cvtsepi32_epi16( __m128i a);
VPMOVDW __m128i __mm256_mask_cvtepi32_epi16( __m128i s, __mmask8 k, __m128i a);
VPMOVDW __m128i __mm256_maskz_cvtepi32_epi16( __mmask8 k, __m128i a);
VPMOVDW __m128i __mm256_maskzword_cvtepi32_epi16( __m128i s, __mmask8 k, __m128i a);
VPMOVDW __m128i __mm_cvtepi32_epi16( __m128i a);
VPMOVDW __m128i __mm_mask_cvtepi32_epi16( __m128i s, __mmask8 k, __m128i a);
VPMOVDW __m128i __mm_maskz_cvtepi32_epi16( __mmask8 k, __m128i a);
VPMOVDW __m128i __mm_maskzword_cvtepi32_epi16( __m128i s, __mmask8 k, __m128i a);
VPMOVDW __m256i __mm256_cvtsepi32_epi16( __m256i a);
VPMOVDW __m256i __mm256_mask_cvtepi32_epi16( __m256i s, __mmask16 k, __m256i a);
VPMOVDW __m256i __mm256_maskword_cvtepi32_epi16( __m256i s, __mmask16 k, __m256i a);
VPMOVDW __m256i __mm256_storew_epi16( void * d, __mmask16 k, __m256i 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.
## VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1</td>
<td>RM/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1</td>
<td>RM/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1</td>
<td>RM/V</td>
<td>AVX512Bw</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1</td>
<td>RM/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1</td>
<td>RM/V</td>
<td>AVX512VL AVX512Bw</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 28 /r VPMOVM2W zmm1, k1</td>
<td>RM/V</td>
<td>AVX512Bw</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1</td>
<td>RM/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1</td>
<td>RM/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1</td>
<td>RM/V</td>
<td>AVX512DQ</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1</td>
<td>RM/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1</td>
<td>RM/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W1 38 /r VPMOVM2Q zmm1, k1</td>
<td>RM/V</td>
<td>AVX512DQ</td>
<td>Provides the destination register with all 1's or all 0's based on the corresponding bit in k1.</td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

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

### Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1's or all 0's depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register.

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

VPMOVM2B (EVEX encoded versions)
(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:i] := -1
    ELSE DEST[i+15:i] := 0
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPMOVM2D (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF SRC[j]
    THEN DEST[i+31:i] := -1
    ELSE DEST[i+31:i] := 0
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPMOVM2Q (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF SRC[j]
    THEN DEST[i+63:i] := -1
    ELSE DEST[i+63:i] := 0
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalents

VPMOVM2B __m512i __mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i __mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i __mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i __mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i __mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i __mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i __mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i __mm256_movm_epi16(__mmask16);
VPMOVM2B __m128i __mm_movm_epi8(__mmask16);
VPMOVM2D __m128i __mm_movm_epi32(__mmask8);
VPMOVM2Q __m128i __mm_movm_epi64(__mmask8);
VPMOVM2W __m128i __mm_movm_epi16(__mmask8);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-55, "Type E7NM Class Exception Conditions"; additionally:

#UD If EVEX.vvvv != 1111B.
VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

### Instruction Set Reference, V-Z

**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.

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<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</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</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</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</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</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</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</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</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</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.br/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

**VPMOVQ8 instruction (EVEX encoded versions) when dest is a register**

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

FOR j := 0 TO KL-1
i := j * 8
m := j * 64
IF k1[j] OR *no writemask*
  THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] := 0
  FI;
ENDFOR

**VPMOVQ8 instruction (EVEX encoded versions) when dest is memory**

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

FOR j := 0 TO KL-1
i := j * 8
m := j * 64
IF k1[j] OR *no writemask*
  THEN DEST[i+7:i] := TruncateQuadWordToByte (SRC[m+63:m])
  ELSE
    *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

**VPMOVQ5 instruction (EVEX encoded versions) when dest is a register**

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

FOR j := 0 TO KL-1
i := j * 8
m := j * 64
IF k1[j] OR *no writemask*
  THEN DEST[i+7:i] := SaturateSignedQuadWordToByte (SRC[m+63:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] := 0
  FI;
ENDFOR

DEST[MAXVL-1:VL/8] := 0;

VPMOVQ8/VPMOVQ5/VPMOVQ5—Down Convert Qword to Byte

Vol. 2C  5-443
VPMOVSQB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateSignedQuadWordToByte (SRC[m+63:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR

VPMOVUSQB instruction (EVEX encoded versions) when dest is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedQuadWordToByte (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+7:i] remains unchanged*
      ELSE *zeroing-masking* ; zeroing-masking
        DEST[i+7:i] := 0
      FI
    FI;
ENDFOR

DEST[MAXVL-1:VL/8] := 0;

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 8
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedQuadWordToByte (SRC[m+63:m])
    ELSE
      *DEST[i+7:i] remains unchanged* ; merging-masking
    FI;
ENDFOR
**Intel C/C++ Compiler Intrinsic Equivalents**

VPMOVQB __m128i _mm512_cvtepi64_epi8( __m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8( __m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_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 __m128i _mm512_mask_cvtepi64_epi8( __m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_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 __m128i _mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB void _mm512_maskz_cvtepi64_epi8( void * d, __mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_cvtepi64_epi8( __m128i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8( __mmask8 k, __m128i a);
VPMOVQB void _mm512_maskz_cvtepi64_epi8( void * d, __mmask8 k, __m128i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m128i a);
VPMOVQB __m128i _mm128_cvtepi64_epi8( __m128i a);
VPMOVQB __m128i _mm_maskz_cvtepi64_epi8( __mmask8 k, __m128i a);
VPMOVQB void _mm_maskz_cvtepi64_epi8( void * d, __mmask8 k, __m128i a);
VPMOVQB void _mm_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m128i a);
VPMOVQB __m128i _mm256_cvtepi64_epi8( __m256i a);
VPMOVQB __m128i _mm256_maskz_cvtepi64_epi8( __mmask8 k, __m256i a);
VPMOVQB void _mm256_maskz_cvtepi64_epi8( void * d, __mmask8 k, __m256i a);
VPMOVQB void _mm256_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m256i 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.
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.F3.0F38.W0 35 /r</td>
<td>A</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>
<td></td>
</tr>
<tr>
<td>VPMOVQD xmm1/m128 {k1}[z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 25 /r</td>
<td>A</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>
<td></td>
</tr>
<tr>
<td>VPMOVSQD xmm1/m64 {k1}[z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 15 /r</td>
<td>A</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>
<td></td>
</tr>
<tr>
<td>VPMOVUSQD xmm1/m64 {k1}[z], xmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 35 /r</td>
<td>A</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>
<td></td>
</tr>
<tr>
<td>VPMOVQD ymm1/m128 {k1}[z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 25 /r</td>
<td>A</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>
<td></td>
</tr>
<tr>
<td>VPMOVSQD ymm1/m128 {k1}[z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 15 /r</td>
<td>A</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>
<td></td>
</tr>
<tr>
<td>VPMOVUSQD ymm1/m128 {k1}[z], ymm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 35 /r</td>
<td>A</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>
<td></td>
</tr>
<tr>
<td>VPMOVQD ymm1/m256 {k1}[z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 25 /r</td>
<td>A</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>
<td></td>
</tr>
<tr>
<td>VPMOVSQD ymm1/m256 {k1}[z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 15 /r</td>
<td>A</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>
<td></td>
</tr>
<tr>
<td>VPMOVUSQD ymm1/m256 {k1}[z], zmm2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask.

Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

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

**VPMOVQD instruction (EVEX encoded version) reg-reg form**

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

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

DEST[MAXVL-1:VL/2] := 0;

**VPMOVQD instruction (EVEX encoded version) memory form**

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

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

**VPMOVVSQD instruction (EVEX encoded version) reg-reg form**

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

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

DEST[MAXVL-1:VL/2] := 0;
VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

VPMOVQD/VPMOVSQD/VPMOVUSQD instruction (EVEX encoded version) memory form

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

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

\[i := j \times 32\]

\[m := j \times 64\]

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

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

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

\[i := j \times 32\]

\[m := j \times 64\]

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

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

ELSE

IF *merging-masking* ; merging-masking

THEN *\(DEST[i+31:i]\) remains unchanged* ; \(DEST[i+31:i] := 0\)

ELSE *zeroing-masking* ; zeroing-masking

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

FI

FI;

ENDFOR

\(DEST[MAXVL-1:VL/2] := 0;\)

VPMOVQD/VPMOVSQD/VPMOVUSQD instruction (EVEX encoded version) memory form

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

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

\[i := j \times 32\]

\[m := j \times 64\]

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

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

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 s, __mmask8 k, __m512i a);
VPMOVQD void _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m512i a);
VPMOVQD __m256i __m512i_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __m512i_mask_cvtepi64_epi32(__m512i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __m512i_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD _mm512_cvtepi64_storeu_epi32(void * d, __mmask8 k __m512i a);
VPMOVQD __m256i __m512i_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __m512i_mask_cvtepi64_epi32(__m512i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __m512i_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m512i a);
VPMOVQD __m256i __m512i_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __m512i_mask_cvtepi64_epi32(__m512i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __m512i_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m512i a);
VPMOVQD __m256i __m512i_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __m512i_mask_cvtepi64_epi32(__m512i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __m512i_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m512i a);
VPMOVQD __m256i __m512i_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i __m512i_mask_cvtepi64_epi32(__m512i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __m512i_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m512i a);
VPMOVQD __m128i __m256i_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i __m256i_mask_cvtepi64_epi32(__m256i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD __m128i __m256i_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m128i b);
VPMOVQD __m128i __m256i_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i __m256i_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m128i b);
VPMOVQD __m128i __m256i_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i __m256i_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m128i b);
VPMOVQD __m128i __m256i_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i __m256i_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m128i b);
VPMOVQD __m128i __m256i_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i __m256i_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m128i b);
VPMOVQD __m128i __m256i_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i __m256i_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k __m128i b);
VPMOVQD __m128i __m256i_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i __m256i_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD __m128i __m256i_mask_cvtepi64_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

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<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 34 1r VPMOVQW xmm1/m32 [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 word integers in xmm1/m32 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.F3.0F38.W0 24 1r VPMOVSQW xmm1/m32 [k1][z], xmm2</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 1r VPMOVSQW xmm1/m32 [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 word integers in xmm1/m32 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 34 1r VPMOVQW xmm1/m64 [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 word integers in xmm1/m64 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 24 1r VPMOVSQW xmm1/m64 [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 word integers in xmm1/m64 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.F3.0F38.W0 14 1r VPMOVSQW xmm1/m64 [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 word integers in xmm1/m64 using unsigned saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 34 1r VPMOVQW xmm1/m128 [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 word integers in xmm1/m128 with truncation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 24 1r VPMOVSQW xmm1/m128 [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 word integers in xmm1/m128 using signed saturation under writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.F3.0F38.W0 14 1r VPMOVSQW xmm1/m128 [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 word integers in xmm1/m128 using unsigned saturation 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>Quarter Mem</td>
<td>ModRM:r/m (w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

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

**VPMOVQW instruction (EVEX encoded versions) when dest is a register**

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

FOR \(j := 0 \text{ TO } KL-1\)
  \(i := j \times 16\)
  \(m := j \times 64\)
  IF \(k1[j] \text{ OR *no writemask*}\)
    THEN \(\text{DEST}[i+15:i] := \text{TruncateQuadWordToWord}(\text{SRC}[m+63:m])\)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST*[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      \(\text{DEST}[i+15:i] := 0\)
    FI
  FI;
ENDFOR
\(\text{DEST}[\text{MAXVL}-1:\text{VL}/4] := 0;\)

**VPMOVQW instruction (EVEX encoded versions) when dest is memory**

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

FOR \(j := 0 \text{ TO } KL-1\)
  \(i := j \times 16\)
  \(m := j \times 64\)
  IF \(k1[j] \text{ OR *no writemask*}\)
    THEN \(\text{DEST}[i+15:i] := \text{TruncateQuadWordToWord}(\text{SRC}[m+63:m])\)
  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 \(j := 0 \text{ TO } KL-1\)
  \(i := j \times 16\)
  \(m := j \times 64\)
  IF \(k1[j] \text{ OR *no writemask*}\)
    THEN \(\text{DEST}[i+15:i] := \text{SaturateSignedQuadWordToWord}(\text{SRC}[m+63:m])\)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST*[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      \(\text{DEST}[i+15:i] := 0\)
    FI
  FI;
ENDFOR
\(\text{DEST}[\text{MAXVL}-1:\text{VL}/4] := 0;\)
VPMOVQSw 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

VPMOVUSQSw 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;

VPMOVUSQSw instruction (EVEX encoded versions) when dest is memory
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
  THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
**Intel C/C++ Compiler Intrinsic Equivalents**

VPMOVQW __m128i _mm512_cvtepi64_epi16( __m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16( __m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16( __mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVWSQW __m128i _mm512_cvtsepi64_epi16( __m512i a);
VPMOVWSQW __m128i _mm512_mask_cvtsepi64_epi16( __m512i s, __mmask8 k, __m512i a);
VPMOVWSQW __m128i _mm512_maskz_cvtsepi64_epi16( __mmask8 k, __m512i a);
VPMOVWSQW void _mm512_mask_cvtsepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_cvtusepi64_epi16( __m512i a);
VPMOVUSQW __m128i _mm512_mask_cvtusepi64_epi16( __m512i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_maskz_cvtusepi64_epi16( __mmask8 k, __m512i a);
VPMOVUSQW void _mm512_mask_cvtusepi64_storeu_epi16( void * d, __mmask8 k, __m512i a);

**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

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

\(i := j \times 8\)

\(m := j \times 16\)

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

THEN \(\text{DEST}[i+7:i] := \text{TruncateWordToByte}(\text{SRC}[m+15:m])\)

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;

ENDFOR

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

VPMOVWB instruction (EVEX encoded versions) when dest is memory

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

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

\(i := j \times 8\)

\(m := j \times 16\)

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

THEN \(\text{DEST}[i+7:i] := \text{TruncateWordToByte}(\text{SRC}[m+15:m])\)

ELSE

*DEST*[i+7:i] remains unchanged* ; merging-masking

FI;

ENDFOR

VPMOVSwB instruction (EVEX encoded versions) when dest is a register

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

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

\(i := j \times 8\)

\(m := j \times 16\)

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

THEN \(\text{DEST}[i+7:i] := \text{SaturateSignedWordToByte}(\text{SRC}[m+15:m])\)

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;

ENDFOR

\(\text{DEST}[\text{MAXVL-1:VL/2}] := 0;\)
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
ENDIF
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;
ENDIF
**Intel C/C++ Compiler Intrinsic Equivalents**

VPMOVUSWB __m256i_mm512_cvtepi16_epi8(__m512i a);
VPMOVUSWB __m256i_mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB __m256i_mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSWB void_mm512_mask_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVUSWB __m256i_mm512_cvtepi16_epi8( __m512i a);
VPMOVUSWB __m256i_mm512_maskz_cvtepi16_epi8( __mmask32 k, __m512i b);
VPMOVUSWB void_mm512_mask_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);
VPMOVUSWB __m256i_mm512_cvtepi16_epi8( __m512i a);
VPMOVUSWB __m256i_mm512_maskz_cvtepi16_epi8( __mmask32 k, __m512i b);
VPMOVUSWB void_mm512_mask_cvtepi16_storeu_epi8(void *, __mmask32 k, __m512i b);

VPMOVUSWB __m128i_mm256_cvtepi16_epi8(__m256i a);
VPMOVUSWB __m128i_mm256_mask_cvtepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i_mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void_mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m128i_mm256_cvtepi16_epi8(__m256i a);
VPMOVUSWB __m128i_mm256_mask_cvtepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i_mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void_mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m128i_mm256_cvtepi16_epi8(__m256i a);
VPMOVUSWB __m128i_mm256_mask_cvtepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i_mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void_mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 k, __m256i b);
VPMOVUSWB __m128i_mm256_cvtepi16_epi8(__m256i a);
VPMOVUSWB __m128i_mm256_mask_cvtepi16_epi8(__m256i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i_mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void_mm256_mask_cvtepi16_storeu_epi8(void *, __mmask16 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.
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>

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>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;
    ENDIF
  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 _mm128i_multishift_epi64_epi8(_m128i a, _m128i b);
VPMULTISHIFTQB _m128i _mm128i_mask_multishift_epi64_epi8(_m128i s, _mmask8 k, _m128i a, _m128i b);
VPMULTISHIFTQB _m128i _mm128i_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</td>
<td>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>
</tr>
<tr>
<td>VPOPCNTB xmm1{k1}{z}, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 54/r</td>
<td>A</td>
<td>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>
</tr>
<tr>
<td>VPOPCNTB ymm1{k1}{z}, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 54/r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG</td>
<td>Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>VPOPCNTB zmm1{k1}{z}, zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 54/r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG</td>
<td>Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>VPOPCNTW xmm1{k1}{z}, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 54/r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG</td>
<td>Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>VPOPCNTW ymm1{k1}{z}, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 54/r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512_BITALG</td>
<td>Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>VPOPCNTW zmm1{k1}{z}, zmm2/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 55/r</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>VPOPCNTD xmm1{k1}{z}, xmm2/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 55/r</td>
<td>B</td>
<td>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>
</tr>
<tr>
<td>VPOPCNTD ymm1{k1}{z}, ymm2/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 55/r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ</td>
<td>Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>VPOPCNTD zmm1{k1}{z}, zmm2/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 55/r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ</td>
<td>Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1.</td>
</tr>
<tr>
<td>VPOPCNTQ xmm1{k1}{z}, xmm2/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 55/r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ</td>
<td>Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1.</td>
</tr>
<tr>
<td>VPOPCNTQ ymm1{k1}{z}, ymm2/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 55/r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512_VPOPCNTDQ</td>
<td>Counts the number of bits set to one in zmm2/m512/m64bcst and puts the result in zmm1 with writemask k1.</td>
</tr>
<tr>
<td>VPOPCNTQ zmm1{k1}{z}, zmm2/m512/m64bcst</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 Mem</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>B</td>
<td>Full</td>
<td>ModRMreg (w)</td>
<td>ModRM:r/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);

VPOPCNTW __m128i _mm_popcnt_epi64(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi64(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi64(__mmask8, __m128i);
VPOPCNTQ __m256i _mm256_popcnt_epi64(__m256i);
VPOPCNTQ __m256i _mm256_mask_popcnt_epi64(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i _mm256_maskz_popcnt_epi64(__mmask8, __m256i);
VPOPCNTQ __m512i _mm512_popcnt_epi64(__m512i);
VPOPCNTQ __m512i _mm512_mask_popcnt_epi64(__m512i, __mmask8, __m512i);
VPOPCNTQ __m512i _mm512_maskz_popcnt_epi64(__mmask8, __m512i);

VPOPCNTQ __m256i _mm256_popcnt_epi32(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i _mm256_mask_popcnt_epi32(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i _mm256_maskz_popcnt_epi32(__mmask8, __m256i);
VPOPCNTQ __m512i _mm512_popcnt_epi32(__m512i, __mmask8, __m512i);
VPOPCNTQ __m512i _mm512_mask_popcnt_epi32(__m512i, __mmask8, __m512i);
VPOPCNTQ __m512i _mm512_maskz_popcnt_epi32(__mmask8, __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, __mmask8, __m256i);
VPOPCNTB __m256i _mm256_mask_popcnt_epi8(__m256i, __mmask32, __m256i);
VPOPCNTB __m256i _mm256_maskz_popcnt_epi8(__mmask32, __m256i);
VPOPCNTB __m512i _mm512_popcnt_epi8(__m512i, __mmask64, __m512i);
VPOPCNTB __m512i _mm512_mask_popcnt_epi8(__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 VPROLVD xmm1[k1][2], 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.128.66.0F38.W0 72 /1 ib VPROLD xmm1 [k1][x], 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][2], 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.128.66.0F38.W1 72 /1 ib VPROLQ xmm1 [k1][x], 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 VPROLVD ymm1[k1][2], 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.256.66.0F38.W1 15 /r VPROLVQ ymm1[k1][2], ymm2, ymm3/m256/m64bcst, imm8</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.512.66.0F38.W0 15 /r VPROLVD zmm1[k1][2], 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.512.66.0F38.W1 72 /1 ib VPROLQ zmm1 [k1][x], zmm2/m512/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512F</td>
<td>Rotate left of doublewords in zmm2/m512/m64bcst 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][x], 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.512.66.0F38.W0 15 /r VPROLQ zmm1 [k1][x], 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:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPROLQ (EVEX encoded versions)

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

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[63:0], imm8)
      ELSE DEST[i+63:i] := LEFT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPROLVQ (EVEX encoded versions)

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

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

Intel C/C++ Compiler Intrinsic Equivalent

VPROLD __m512i _mm512_rol_epi32(__m512i a, int imm);
VPROLD __m512i _mm512_mask_rol_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPROLD __m512i _mm512_maskz_rol_epi32(__mmask16 k, __m512i a, int imm);
VPROLD __m256i _mm256_rol_epi32(__m256i a, int imm);
VPROLD __m256i _mm256_mask_rol_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLD __m256i _mm256_maskz_rol_epi32(__mmask8 k, __m256i a, int imm);
VPROLD __m128i _mm_rol_epi32(__m128i a, int imm);
VPROLD __m128i _mm_mask_rol_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLD __m128i _mm_maskz_rol_epi32(__mmask8 k, __m128i a, int imm);
VPROLD __m512i _mm512_rolv_epi32(__m512i a, __m512i cnt);
VPROLD __m512i _mm512_mask_rolv_epi32(__mmask16 k, __m512i a, __m512i b, __m512i cnt);
VPROLD __m512i _mm512_maskz_rolv_epi32(__mmask16 k, __m512i a, __m512i b, __m512i cnt);
VPROLD __m256i _mm256_rolv_epi32(__m256i a, __m256i b, __m256i cnt);
VPROLD __m256i _mm256_mask_rolv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLD __m256i _mm256_maskz_rolv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPROLD __m128i _mm_rolv_epi32(__m128i a, __m128i b, __m128i cnt);
VPROLD __m128i _mm_mask_rolv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLD __m128i _mm_maskz_rolv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPROLD __m512i _mm512_rolv_epi64(__m512i a, __m512i cnt);
VPROLD __m512i _mm512_mask_rolv_epi64(__mmask8 k, __m512i a, __m512i b, __m512i cnt);
VPROLD __m512i _mm512_maskz_rolv_epi64(__mmask8 k, __m512i a, __m512i b, __m512i cnt);
VPROLD __m256i _mm256_rolv_epi64(__m256i a, __m256i b, __m256i cnt);
VPROLD __m256i _mm256_mask_rolv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLD __m256i _mm256_maskz_rolv_epi64(__mmask8 k, __m256i a, __m256i cnt)

VPROLVQ __m128i __mm_mask_rolv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVQ __m128i __mm_maskz_rolv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions
None

Other Exceptions
EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions".
### 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</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W0 72 /0 lb</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 14 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F:W1 72 /0 lb</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 14 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W0 72 /0 lb</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 14 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F:W1 72 /0 lb</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 14 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W0 72 /0 lb</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 14 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F:W1 72 /0 lb</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.</td>
</tr>
</tbody>
</table>

#### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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

\[
\text{RIGHT\_ROTATE\_DWORDS}(\text{SRC}, \text{COUNT\_SRC})
\]
\[
\text{COUNT} := \text{COUNT\_SRC} \mod 32;
\]
\[
\text{DEST}[31:0] := (\text{SRC} \gg \text{COUNT}) | (\text{SRC} \ll (32 - \text{COUNT}));
\]

\[
\text{RIGHT\_ROTATE\_QWORDS}(\text{SRC}, \text{COUNT\_SRC})
\]
\[
\text{COUNT} := \text{COUNT\_SRC} \mod 64;
\]
\[
\text{DEST}[63:0] := (\text{SRC} \gg \text{COUNT}) | (\text{SRC} \ll (64 - \text{COUNT}));
\]

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
\[
\text{IF (EVEX.b = 1) AND (\text{SRC1 *is memory*})}
\]
\[
\text{THEN \text{DEST}[i+31:i] := RIGHT\_ROTATE\_DWORDS( \text{SRC1}[31:0], \text{imm8})}
\]
\[
\text{ELSE \text{DEST}[i+31:i] := RIGHT\_ROTATE\_DWORDS(\text{SRC1}[i+31:i], \text{imm8})}
\]
\[
\text{FI};
\]
ELSE
\[
\text{IF *merging-masking* ; merging-masking}
\]
\[
\text{THEN \text{*DEST[i+31:i] remains unchanged*}
\]
\[
\text{ELSE \text{*zeroing-masking* ; zeroing-masking}
\]
\[
\text{DEST[i+31:i] := 0}
\]
\[
\text{FI}
\]
ENDIF

DEST[MAXVL-1:VL] := 0
VPRORVD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR "no writemask" THEN
    IF (EVEX.b = 1) AND (SRC2 "is memory")
      THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
      ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
    FI;
  ELSE
    IF "merging-masking" THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking" THEN DEST[i+31:i] := 0
tend IF;
ENDFOR
DEST[MAXVL-1:VL] := 0

VPRORQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR "no writemask" THEN
    IF (EVEX.b = 1) AND (SRC1 "is memory")
      THEN DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[63:0], imm8)
      ELSE DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
    FI;
  ELSE
    IF "merging-masking" THEN *DEST[i+63:i] remains unchanged*
    ELSE *zeroing-masking" THEN DEST[i+63:i] := 0
tend IF;
ENDFOR
DEST[MAXVL-1:VL] := 0
VPRORVQ (EVEX encoded versions)

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

FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN DEST[i+63:i] := RIGHT Rotate QWords(SRC1[i+63:i], SRC2[63:0])
ELSE DEST[i+63:i] := RIGHT Rotate QWords(SRC1[i+63:i], SRC2[i+63:i])
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE *zeroing-masking* ; zeroing-masking
DEST[i+63:i] := 0
FI
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_rorv_epi32(__m512i a, __m512i cnt);
VPRORD __m512i _mm512_mask_rorv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPRORD __m512i _mm512_maskz_rorv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPRORD __m256i _mm256_rorv_epi32(__m256i a, __m256i cnt);
VPRORD __m256i _mm256_mask_rorv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORD __m256i _mm256_maskz_rorv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPRORD __m128i _mm_rorv_epi32(__m128i a, __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 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 __m256i _mm256_rorv_epi64(__m256i a, __m256i cnt);
VPRORD __m256i _mm256_mask_rorv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORD __m256i _mm256_maskz_rorv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPRORD __m128i _mm_rorv_epi64(__m128i a, __m128i cnt);
VPRORD __m128i _mm_mask_rorv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORD __m128i _mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i cnt);
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

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 * 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
ENDFOR
k1[MAX_KL-1:KL] := 0

VPSCATTERQQ (EVEX encoded versions)
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR "no writemask"
    THEN MEM[BASE_ADDR + (VINDEX[j+63:j]) * SCALE + DISP] := SRC[i+63:i]
  ENDIF
ENDFOR
k1[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VPSCATTERDD void _mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void _mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_i32scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);

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 VPSHLDW xmm1{k1}{z}, xmm2, xmm3/m128, imm8</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 VPSHLDW ymm1{k1}{z}, ymm2, ymm3/m256, imm8</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 VPSHLDW zmm1{k1}{z}, zmm2, zmm3/m512, imm8</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 /ib VPSHLDD xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8</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 /ib VPSHLDD ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</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 /ib VPSHLDD zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8</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 zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 71 /r /ib VPSHLDQ xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8</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 /ib VPSHLDQ ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8</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 /ib VPSHLDQ zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8</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 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.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.
**Instruction Set Reference, V-Z**

**Operation**

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

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

**VPSHLDQ** DEST, SRC2, SRC3, imm8  
(KL, VL) = (2, 128), (4, 256), (8, 512)  
FOR j := 0 TO KL-1:  
  IF SRC3 is broadcast memop:  
    tsrc3 := SRC3.qword[0]  
  ELSE:  
    tsrc3 := SRC3.qword[j]  
  IF MaskBit(j) OR *no writemask*:  
    tmp := concat(SRC2.qword[j], tsrc3) \(<< (\text{imm8} \& 63)\)  
    DEST.qword[j] := tmp.qword[1]  
  ELSE IF *zeroing*:  
    DEST.qword[j] := 0  
  *ELSE DEST.qword[j] remains unchanged*  
DEST[MAX_VL-1:VL] := 0
Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Type E4.
## VPSHLDV — Concatenate and Variable Shift Packed Data Left Logical

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<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 VPSHLDVQ 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 VPSHLDVQ 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 VPSHLDVQ 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>
</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
The VPSHLDVQ instruction in Intel's instruction set reference manual is explained as follows:

**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**

VPSHLDVQ m128i _mm512_shldv_epi64(m128i, m128i, m128i);
VPSHLDVQ m128i _mm512_mask_shldv_epi64(m128i, mmask8, m128i, m128i);
VPSHLDVQ m128i _mm512_maskz_shldv_epi64(mmask8, m128i, m128i, m128i);
VPSHLDVQ m256i _mm256_shldv_epi16(m256i, m256i, m256i);
VPSHLDVQ m256i _mm256_mask_shldv_epi16(m256i, mmask16, m256i, m256i);
VPSHLDVQ m256i _mm256_maskz_shldv_epi16(mmask16, m256i, m256i, m256i);
VPSHLDVQ m512i _mm512_shldv_epi64(m512i, m512i, m512i);
VPSHLDVQ m512i _mm512_mask_shldv_epi64(m512i, mmask8, m512i, m512i);
VPSHLDVQ m512i _mm512_maskz_shldv_epi64(mmask8, m512i, m512i, m512i);

**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 AVX512VL</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.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 right by constant value. This instruction supports memory fault suppression.
**VPSHRD** — Concatenate and Shift Packed Data Right Logical

**Operation**

**VPSHRDW** DEST, SRC2, SRC3, imm8

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

FOR j := 0 TO KL-1:
  IF MaskBit(j) OR *no writemask*:
    DEST.word[j] := concat(SRC3.word[j], SRC2.word[j]) >> (imm8 & 15)
  ELSE IF *zeroing*:
    DEST.word[j] := 0
  *ELSE DEST.word[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

**VPSHRDD** DEST, SRC2, SRC3, imm8

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

FOR j := 0 TO KL-1:
  IF SRC3 is broadcast memop:
    tsrc3 := SRC3.dword[0]
  ELSE:
    tsrc3 := SRC3.dword[j]
  IF MaskBit(j) OR *no writemask*:
    DEST.dword[j] := concat(tsrc3, SRC2.dword[j]) >> (imm8 & 31)
  ELSE IF *zeroing*:
    DEST.dword[j] := 0
  *ELSE DEST.dword[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

**VPSHRDQ** DEST, SRC2, SRC3, imm8

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

FOR j := 0 TO KL-1:
  IF SRC3 is broadcast memop:
    tsrc3 := SRC3.qword[0]
  ELSE:
    tsrc3 := SRC3.qword[j]
  IF MaskBit(j) OR *no writemask*:
    DEST.qword[j] := concat(tsrc3, SRC2.qword[j]) >> (imm8 & 63)
  ELSE IF *zeroing*:
    DEST.qword[j] := 0
  *ELSE DEST.qword[j] remains unchanged*

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

VPSHRDQ __m128i __mm_shrdi_epi64(__m128i, __m128i, int);
VPSHRDQ __m128i __mm_mask_shrdi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDQ __m128i __mm_maskz_shrdi_epi64(__mmask8, __m128i, __m128i, int);
VPSHRDQ __m256i __mm256_shrdi_epi64(__m256i, __m256i, int);
VPSHRDQ __m256i __mm256_mask_shrdi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDQ __m256i __mm256_maskz_shrdi_epi64(__mmask8, __m256i, __m256i, int);
VPSHRDQ __m512i __mm512_shrdi_epi64(__m512i, __m512i, int);
VPSHRDQ __m512i __mm512_mask_shrdi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHRDQ __m512i __mm512_maskz_shrdi_epi64(__mmask8, __m512i, __m512i, int);
VPSHRDD __m128i __mm_shrdi_epi32(__m128i, __m128i, int);
VPSHRDD __m128i __mm_mask_shrdi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDD __m128i __mm_maskz_shrdi_epi32(__mmask8, __m128i, __m128i, int);
VPSHRDD __m256i __mm256_shrdi_epi32(__m256i, __m256i, int);
VPSHRDD __m256i __mm256_mask_shrdi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHRDD __m256i __mm256_maskz_shrdi_epi32(__mmask8, __m256i, __m256i, int);
VPSHRDD __m512i __mm512_shrdi_epi32(__m512i, __m512i, int);
VPSHRDD __m512i __mm512_mask_shrdi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHRDD __m512i __mm512_maskz_shrdi_epi32(__mmask16, __m512i, __m512i, int);
VPSHRDW __m128i __mm_shrdi_epi16(__m128i, __m128i, int);
VPSHRDW __m128i __mm_mask_shrdi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHRDW __m128i __mm_maskz_shrdi_epi16(__mmask8, __m128i, __m128i, int);
VPSHRDW __m256i __mm256_shrdi_epi16(__m256i, __m256i, int);
VPSHRDW __m256i __mm256_mask_shrdi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHRDW __m256i __mm256_maskz_shrdi_epi16(__mmask16, __m256i, __m256i, int);
VPSHRDW __m512i __mm512_shrdi_epi16(__m512i, __m512i, int);
VPSHRDW __m512i __mm512_mask_shrdi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHRDW __m512i __mm512_maskz_shrdi_epi16(__mmask32, __m512i, __m512i, int);

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Type E4.
VPSHRDV — Concatenate and Variable Shift Packed Data Right Logical

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<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 VPSHRDVW 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 right by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 72 /r VPSHRDVW 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 right by value in ymm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 72 /r VPSHRDVW 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 right by value in zmm3/m512 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 73 /r VPSHRDVQ 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 right by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 73 /r VPSHRDVQ 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 right by value in ymm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 73 /r VPSHRDVQ 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 right by value in zmm3/m512 into zmm1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 73 /r VPSHRDVQ 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 right by value in xmm3/m128 into xmm1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 73 /r VPSHRDVQ 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 right by value in ymm3/m256 into ymm1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 73 /r VPSHRDVQ 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 right by value in zmm3/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 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

**VPSHRDVQ DEST, SRC2, SRC3**

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

FOR j := 0 TO KL-1:
   IF SRC3 is broadcast memop:
      tsrc3 := SRC3.dword[0]
   ELSE:
      tsrc3 := SRC3.dword[j]
   IF MaskBit(j) OR *no writemask*:
      DEST.dword[j] := concat(SRC2.dword[j], DEST.dword[j]) >> (tsrc3 & 31)
   ELSE IF *zeroing*:
      DEST.dword[j] := 0
   *ELSE DEST.dword[j] remains unchanged*

DEST[MAX_VL-1:VL] := 0

**VPSHRDVQ DEST, SRC2, SRC3**

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

FOR j := 0 TO KL-1:
   IF SRC3 is broadcast memop:
      tsrc3 := SRC3.qword[0]
   ELSE:
      tsrc3 := SRC3.qword[j]
   IF MaskBit(j) OR *no writemask*:
      DEST.qword[j] := concat(SRC2.qword[j], DEST.qword[j]) >> (tsrc3 & 63)
   ELSE IF *zeroing*:
      DEST.qword[j] := 0
   *ELSE DEST.qword[j] remains unchanged*

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

VPSHRDVQ __m128i _mm_shrdv_epi64(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i _mm_mask_shrdv_epi64(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i _mm_maskz_shrdv_epi64(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i _mm256_shrdv_epi64(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_mask_shrdv_epi64(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_maskz_shrdv_epi64(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i _mm512_shrdv_epi64(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_mask_shrdv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_maskz_shrdv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHRDVQ __m128i _mm_shrdv_epi32(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i _mm_mask_shrdv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i _mm_maskz_shrdv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i _mm256_shrdv_epi32(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_mask_shrdv_epi32(__m256i, __mmask16, __m256i, __m256i);
VPSHRDVQ __m256i _mm256_maskz_shrdv_epi32(__mmask16, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i _mm512_shrdv_epi32(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_mask_shrdv_epi32(__m512i, __mmask32, __m512i, __m512i);
VPSHRDVQ __m512i _mm512_maskz_shrdv_epi32(__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</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</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</td>
<td>ModRM:reg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

Description

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

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

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFBITQMB __mmask16 __mm_bitshuffle_epi64_mask(__m128i, __m128i);
VPSHUFBITQMB __mmask16 __mm_mask_bitshuffle_epi64_mask(__mmask16, __m128i, __m128i);
VPSHUFBITQMB __mmask32 __mm256_bitshuffle_epi64_mask(__m256i, __m256i);
VPSHUFBITQMB __mmask32 __mm256_mask_bitshuffle_epi64_mask(__mmask32, __m256i, __m256i);
VPSHUFBITQMB __mmask64 __mm512_bitshuffle_epi64_mask(__m512i, __m512i);
VPSHUFBITQMB __mmask64 __mm512_mask_bitshuffle_epi64_mask(__mmask64, __m512i, __m512i);
### 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 VPSLLVQ 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 VPSLLVQ 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.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 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-mask* ; merging-mask
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-mask
        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;
ENDIF
IF COUNT_1 < 64 THEN
DEST[127:64] := ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] := 0;
ENDIF
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;
ENDIF
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] := ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] := 0;
ENDIF
DEST[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[i] OR *no writemask* THEN
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[i+63:i] remains unchanged*
ELSE ; zeroing-masking
DEST[i+63:i] := 0
FI
ENDIF
**Intel C/C++ Compiler Intrinsic Equivalent**

VPSLLW  
\[
\text{VPSLLW } \_\_m512i \_\_mm512\_sllv\_epi16(\_\_m512i a, \_\_m512i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m512i \_\_mm512\_mask\_sllv\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m512i \_\_mm512\_maskz\_sllv\_epi16(\_\_m516 k, \_\_m512i a, \_\_m512i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m256i \_\_mm256\_mask\_sllv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m256i \_\_mm256\_maskz\_sllv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m128i \_\_mm128\_mask\_sllv\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m128i \_\_mm128\_maskz\_sllv\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m256i \_\_mm256\_mask\_sllv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m256i \_\_mm256\_maskz\_sllv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m128i \_\_mm128\_mask\_sllv\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m128i \_\_mm128\_maskz\_sllv\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m256i \_\_mm256\_mask\_sllv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m256i \_\_mm256\_maskz\_sllv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m128i \_\_mm128\_mask\_sllv\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m128i \_\_mm128\_maskz\_sllv\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m256i \_\_mm256\_mask\_sllv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m256i \_\_mm256\_maskz\_sllv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m128i \_\_mm128\_mask\_sllv\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVW  
\[
\text{VPSLLVW } \_\_m128i \_\_mm128\_maskz\_sllv\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m512i \_\_mm512\_sllv\_epi32(\_\_m512i a, \_\_m512i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m512i \_\_mm512\_mask\_sllv\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m512i \_\_mm512\_maskz\_sllv\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m256i \_\_mm256\_mask\_sllv\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m256i \_\_mm256\_maskz\_sllv\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m128i \_\_mm128\_mask\_sllv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m128i \_\_mm128\_maskz\_sllv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m256i \_\_mm256\_mask\_sllv\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m256i \_\_mm256\_maskz\_sllv\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m256i \_\_mm256\_sllv\_epi32(\_\_m256i m, \_\_m256i count)
\]

VPSLLVD  
\[
\text{VPSLLVD } \_\_m256i \_\_mm256\_sllv\_epi32(\_\_m256i m, \_\_m256i count)
\]

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VE4-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions”.

EV4-encoded VPSLLVW, see Table 2-29, “Type E4 Class Exception Conditions”.

EV4-encoded VPSLLVW, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions”.
### VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.128.66.0F38.W0 46 /r VPSRAVQ 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 sign bits.</td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 46 /r VPSRAVQ 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 sign bits.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 11 /r VPSRAVW 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 sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 11 /r VPSRAW 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 sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 11 /r VPSRAVW 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 sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 46 /r VPSRAVD 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 sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 46 /r VPSRAVD 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 sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 46 /r VPSRAVD 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 sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 11 /r VPSRAVD 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 sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 11 /r VPSRAVD 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 sign bits using writemask k1.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 11 /r VPSRAVQ 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 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>ModRMr/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>ModRMr/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>ModRMr/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)

FOR j := 0 TO KL-1
   i := j * 16
   IF k1[j] OR *no writemask*
      THEN
         COUNT := SRC2[i+3:i]
         IF COUNT < 16
            THEN DEST[i+15:i] := SignExtend(SRC1[i+15:i] >> COUNT)
            ELSE
               FOR k := 0 TO 15
                  DEST[i+k] := SRC1[i+15]
               ENDFOR;
         ELSE
            IF *merging-masking*; merging-masking
               THEN *DEST[i+15:i] remains unchanged*
            ELSE ; zeroing-masking
               DEST[i+15:i] := 0
            FI
      FI
   ELSE
      IF *merging-masking*; merging-masking
         THEN *DEST[i+15:i] remains unchanged*
      ELSE ; zeroing-masking
         DEST[i+15:i] := 0
      FI
   ENDIF;
ENDFOR;
DEST[MAXVL-1:VL] := 0;
VPSRAVD (VEX.128 version)
COUNT_0 := SRC2[31 : 0]
     (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[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;
                ENDIF
            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;
                ENDIF
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] := 0
            FI
        FI
    ENDFOR;
    DEST[MAXVL-1:VL] := 0;
VPSRAVQ (EVEX encoded version)

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

 FOR j := 0 TO KL-1
   i := j * 64
   IF k1[j] OR *no writemask* THEN
     IF (EVEX.b = 1) AND (SRC2 *is memory*)
     THEN
       COUNT := SRC2[5:0]
       IF COUNT < 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
     FI;
   ELSE
     IF *merging-masking* ; merging-masking
     THEN  *DEST[63:0] remains unchanged*
     ELSE
       DEST[63:0] := 0 ; zeroing-masking
     FI
   FI;
 ENDFOR;
 DEST[MAXVL-1:VL] := 0;
Intel C/C++ Compiler Intrinsic Equivalent

VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmask8 m, __m256i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i a, __m512i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi32(__m256i s, __mmask16 m, __m256i a, __m512i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi32(__mmask16 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m512i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i a, __m256i 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”.
## 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</td>
<td>A V/V</td>
<td>AVX2</td>
<td></td>
<td>Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLVD xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W1 45 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td></td>
<td>Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLVQ xmm1, xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 45 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td></td>
<td>Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLVD ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W1 45 /r</td>
<td>A V/V</td>
<td>AVX2</td>
<td></td>
<td>Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.</td>
</tr>
<tr>
<td>VPSRLVQ ymm1, ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 10 /r</td>
<td>B V/V</td>
<td>AVX512VL AVX512BW</td>
<td></td>
<td>Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>VPSRLVw xmm1 (k1)[z], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 10 /r</td>
<td>B V/V</td>
<td>AVX512VL AVX512BW</td>
<td></td>
<td>Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>VPSRLVw ymm1 (k1)[z], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 10 /r</td>
<td>B V/V</td>
<td>AVX512BW</td>
<td></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>
</tr>
<tr>
<td>VPSRLVw zmm1 (k1)[z], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 45 /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td></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>
</tr>
<tr>
<td>VPSRLVD xmm1 (k1)[z], xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 45 /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td></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>
</tr>
<tr>
<td>VPSRLVD ymm1 (k1)[z], ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 45 /r</td>
<td>C V/V</td>
<td>AVX512F</td>
<td></td>
<td>Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.</td>
</tr>
<tr>
<td>VPSRLVD zmm1 (k1)[z], zmm2, zmm3/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 45 /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td></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>
</tr>
<tr>
<td>VPSRLVQ xmm1 (k1)[z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 45 /r</td>
<td>C V/V</td>
<td>AVX512VL AVX512F</td>
<td></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>
</tr>
<tr>
<td>VPSRLVQ ymm1 (k1)[z], ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 45 /r</td>
<td>C V/V</td>
<td>AVX512F</td>
<td></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>
</tr>
<tr>
<td>VPSRLVQ zmm1 (k1)[z], zmm2, zmm3/m512/m64bcst</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 (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.vvv (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)

(\(K_L, V_L\) = (8, 128), (16, 256), (32, 512))

\[
\text{FOR } j := 0 \text{ TO } K_L - 1 \\
\quad i := j \times 16 \\
\quad \text{IF } k1[j] \text{ OR } \text{*no writemask*} \\
\quad \quad \text{THEN } \text{DEST}[(i+15:i)] := \text{ZeroExtend}(\text{SRC1}[(i+15:i)] \gg \text{SRC2}[(i+15:i)]) \\
\quad \quad \text{ELSE} \\
\quad \quad \quad \text{IF } \text{*merging-masking*} \\
\quad \quad \quad \quad \text{THEN } \text{*DEST}[(i+15:i)] \text{ remains unchanged*} \\
\quad \quad \quad \quad \text{ELSE} \\
\quad \quad \quad \quad \quad \text{DEST}[(i+15:i)] := 0 \\
\quad \quad \text{FI} \\
\quad \text{FI}; \\
\text{ENDFOR;}
\]

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

VPSRLVD (VEX.128 version)

\[
\text{COUNT}_0 := \text{SRC2}[31 : 0] \\
\text{(*) Repeat Each COUNT}_i \text{ for the 2nd through 4th dwords of SRC2*)} \\
\text{COUNT}_3 := \text{SRC2}[127 : 96]; \\
\text{IF COUNT}_0 < 32 \text{ THEN} \\
\quad \text{DEST}[31:0] := \text{ZeroExtend}(\text{SRC1}[31:0] \gg \text{COUNT}_0); \\
\text{ELSE} \\
\quad \text{DEST}[31:0] := 0; \\
\text{(*) Repeat shift operation for 2nd through 4th dwords *)} \\
\text{IF COUNT}_3 < 32 \text{ THEN} \\
\quad \text{DEST}[127:96] := \text{ZeroExtend}(\text{SRC1}[127:96] \gg \text{COUNT}_3); \\
\text{ELSE} \\
\quad \text{DEST}[127:96] := 0; \\
\text{DEST}[\text{MAXVL-1:128}] := 0;
\]
VPSRLVD (VEX.256 version)
COUNT_0 := SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2 *)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
  DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
  DEST[31:0] := 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
  DEST[255:224] := ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
  DEST[255:224] := 0;
DEST[MAXVL-1:256] := 0;

VPSRLVD (EVEX encoded version)
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
      ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] := 0
      FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;

VPSRLVQ (VEX.128 version)
COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64 THEN
  DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
  DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
  DEST[127:64] := ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
  DEST[127:64] := 0;
DEST[MAXVL-1:128] := 0;
VPSRLVQ (VEX.256 version)
COUNT_0 := SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2 *)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] := 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] := ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;

VPSRLVQ (EVEX encoded version)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
      ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
      FI
    FI
ENDFOR;
DEST[MAXVL-1:VL] := 0;
INSTRUCTION SET REFERENCE, V-Z

Intel C/C++ Compiler Intrinsic Equivalent

VPSRLW __m512i __mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLW __m512i __mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLW __m256i __mm256_mask_srlv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLW __m256i __mm256_maskz_srlv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSRLW __m128i __mm_mask_srlv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m128i __mm_maskz_srlv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m256i __mm256_srlv_epi32(__m256i m, __m256i count);
VPSRLVD __m512i __mm512_srlv_epi32(__m512i a, __m512i cnt);
VPSRLVD __m512i __mm512_mask_srlv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m512i __mm512_maskz_srlv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m256i __mm256_mask_srlv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m256i __mm256_maskz_srlv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m512i __mm512_srlv_epi64(__m512i a, __m512i cnt);
VPSRLVQ __m512i __mm512_mask_srlv_epi64(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVQ __m256i __mm256_mask_srlv_epi64(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLVQ __m256i __mm256_maskz_srlv_epi64(__mmask16 k, __m256i a, __m256i cnt);
VPSRLVQ __m128i __mm_mask_srlv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m128i __mm_maskz_srlv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m256i __mm256_srlv_epi64(__m256i m, __m256i count);
VPSRLVD __m128i __mm_srlv_epi32(__m128i a, __m128i cnt);
VPSRLVQ __m128i __mm_srlv_epi64(__m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see 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 VPTERNLOGD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m32bcst as source operands and writing the result to xmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 25 / r ib VPTERNLOGD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m32bcst as source operands and writing the result to ymm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 25 / r ib VPTERNLOGD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A 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>
<td></td>
</tr>
<tr>
<td>EVEX.128.66.0F3A.W1 25 / r ib VPTERNLOGQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m64bcst as source operands and writing the result to xmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 25 / r ib VPTERNLOGQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m64bcst as source operands and writing the result to ymm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 25 / r ib VPTERNLOGQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A 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>
<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>ModRM/r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

Description

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.
Table 5-9 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>Bit(reg1)</th>
<th>Bit(reg2)</th>
<th>Bit(src3)</th>
<th>Bit Result with Imm8=0xE2</th>
<th>Bit(reg1)</th>
<th>Bit(reg2)</th>
<th>Bit(src3)</th>
<th>Bit Result with Imm8=0xE4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<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>
<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>
<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>
<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>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-1 and Table 5-2 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

**Operation**

**VPTERNLOGD (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j * 32

IF k1[j] OR *no writemask*

THEN

FOR k := 0 TO 31

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

THEN DEST[i][k] := imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[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)

FOR j := 0 TO KL-1
i := j * 64
IF k1[j] OR *no writemask*
THEN
FOR k := 0 TO 63
IF (EVEX.b = 1) AND (SRC2 *is memory*)
THEN DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[k]]
ELSE DEST[j][k] := imm[(DEST[i+k] << 2) + (SRC1[i+k] << 1) + SRC2[i+k]]
FI;
ELSE
IF *merging-masking* ; merging-masking
THEN *DEST[63+i:i] remains unchanged*
ELSE ; zeroing-masking
DEST[63+i:i] := 0
FI;
ENDIF;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalents

VPTERNLOGD __m512i _mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_maskz_ternarylogic_epi32(__mmask m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m256i _mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_maskz_ternarylogic_epi32(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m128i _mm_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_mask_ternarylogic_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_maskz_ternarylogic_epi32(__mmask8 m, __m128i a, __m128i b, int imm);

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</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 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>
<td></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 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>
<td></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 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>
<td></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 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>
<td></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.W1 27 /r VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst</td>
<td>B/V/V</td>
<td>AVX512VL 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>
<td></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 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>
<td></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 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>
<td></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 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>
<td></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)

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

Intel C/C++ Compiler Intrinsic Equivalents

VPTESTMB __mmask64 _mm512_test_epi8_mask( __m512i a, __m512i b);
VPTESTMB __mmask64 _mm512_mask_test_epi8_mask( __mmask64, __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_test_epi16_mask( __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_mask_test_epi16_mask( __mmask32, __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_test_epi32_mask( __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_mask_test_epi32_mask( __mmask16, __m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_test_epi64_mask( __m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_mask_test_epi64_mask( __mmask8, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VPTESTMD/Q: See 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>EVX.128.F3.0F38.W0 26 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNMB k2 [k1], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.256.F3.0F38.W0 26 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNMB k2 [k1], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.512.F3.0F38.W0 26 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNMB k2 [k1], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.128.F3.0F38.W1 26 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNW k2 [k1], xmm2, xmm3/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.256.F3.0F38.W1 26 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNW k2 [k1], ymm2, ymm3/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.512.F3.0F38.W1 26 /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512BW</td>
<td>Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNW k2 [k1], zmm2, zmm3/m512</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.128.F3.0F38.W0 27 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNMD k2 [k1], xmm2, xmm3/m128/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.256.F3.0F38.W0 27 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNMD k2 [k1], ymm2, ymm3/m256/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.512.F3.0F38.W0 27 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNMD k2 [k1], zmm2, zmm3/m512/m32bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.128.F3.0F38.W1 27 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNMQ k2 [k1], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.256.F3.0F38.W1 27 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNMQ k2 [k1], ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVX.512.F3.0F38.W1 27 /r</td>
<td>B</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.</td>
</tr>
<tr>
<td>VPTESTNMQ k2 [k1], zmm2, zmm3/m512/m64bcst</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:reg (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:reg (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 \(\text{MaskBit}(j)\) OR *no writemask* THEN

\(\text{DEST}[j] := (\text{SRC1}[i+7:i] \text{ BITWISE AND SRC2}[i+7:i] == 0)? 1 : 0\)

ELSE \(\text{DEST}[j] := 0;\) zeroing masking only

FI

ENDFOR

\(\text{DEST}[\text{MAX_KL-1:KL}] := 0\)

**VPTESTNMW**

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

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

\(i := j*16\)

IF \(\text{MaskBit}(j)\) OR *no writemask* THEN

\(\text{DEST}[j] := (\text{SRC1}[i+15:i] \text{ BITWISE AND SRC2}[i+15:i] == 0)? 1 : 0\)

ELSE \(\text{DEST}[j] := 0;\) zeroing masking only

FI

ENDFOR

\(\text{DEST}[\text{MAX_KL-1:KL}] := 0\)
VPTESTNMD
\( (KL, VL) = (4, 128), (8, 256), (16, 512) \)
FOR \( j := 0 \) TO \( KL-1 \)
\[ i := j \times 32 \]
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 \( KL-1 \)
\[ i := j \times 64 \]
IF MaskBit\( (j) \) OR *no writemask*
    THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST\[j\] := (SRC1\[i+63:i\] BITWISE AND SRC2\[63:0\] == 0)? 1 : 0;
            ELSE DEST\[j\] := (SRC1\[i+63:i\] BITWISE AND SRC2\[i+63:i\] == 0)? 1 : 0;
        FI;
        ELSE DEST\[j\] := 0; zeroing masking only
    FI
ENDFOR
DEST[\( MAX_KL-1:KL \)] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VPTESTNMB __mmask64 _mm512_testn_epi8_mask( __m512i a, __m512i b);
VPTESTNMB __mmask64 _mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 _mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTESTNMB __mmask32 _mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 _mm_testn_epi8_mask(__m128i a, __m128i b);
VPTESTNMB __mmask16 _mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMB __mmask8 _mm512_testn_epi16_mask(__m512i a, __m512i b);
VPTESTNMB __mmask8 _mm512_mask_testn_epi16_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMB __mmask8 _mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTESTNMB __mmask8 _mm_mask_testn_epi16_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMB __mmask8 _mm512_testn_epi32_mask(__m512i a, __m512i b);
VPTESTNMB __mmask8 _mm512_mask_testn_epi32_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMB __mmask8 _mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTESTNMB __mmask8 _mm_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMB __mmask8 _mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTESTNMB __mmask8 _mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMB __mmask8 _mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTESTNMB __mmask8 _mm_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i 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 _mm_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
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](image_url)
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 5-10. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]**

<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 5-11. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS and MAX, MAX_ABS**

<table>
<thead>
<tr>
<th>MIN and MIN_ABS</th>
<th>MAX and MAX_ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1</td>
<td>Src2</td>
</tr>
<tr>
<td>+0</td>
<td>-0</td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
</tr>
</tbody>
</table>

**Table 5-12. Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, (|a| = |b|, a>0, b<0)**

| MIN_ABS (|a| = |b|, a>0, b<0) | MAX_ABS (|a| = |b|, a>0, b<0) |
|-----------|-------------|
| Src1 | Src2 | Result | Src1 | Src2 | Result |
| a   | b     | b      | a   | b     | a      |
| b   | a     | b      | b   | a     | a      |
Operation
RangeDP([SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]])
{
  // Check if SNAN and report IE, see also Table 5-10
  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 (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_maskz_range_pd (__mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_maskz_range_round_pd (__mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d _mm256_range_pd (__m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_maskz_range_pd (__mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d _mm128_range_pd (__m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm128_mask_range_pd (__m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm128_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions".
**VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values**

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W0 50 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 50 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512DQ</td>
<td>Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 50 /r ib</td>
<td>A</td>
<td>V/V</td>
<td>AVX512DQ</td>
<td>Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, 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.ffff (r)</td>
<td>ModRM:r/m (r)</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**Description**

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits 7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.

- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-10. 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(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;
    FI;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[31] << 31) OR (TMP[30:0]); // Preserve Src1 sign bit
    01: dest := TMP[31:0]; // Preserve sign of compare result
    10: dest := (0 << 31) OR (TMP[30:0]); // Zero out sign bit
    11: dest := (1 << 31) OR (TMP[30:0]); // Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
VRANGEPS
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b == 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
      ELSE DEST[i+31:i] := RangeSP (SRC1[i+31:i], SRC2[i+31:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
      FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+31:i] = 0
      FI;
    FI;
  ENDFOR;
DEST[MAXVL-1:VL] := 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between $\pm 150$.

VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;

Where:
- zmm_dst is the destination operand.
- zmm_src is the input operand to compare against $\pm 150$.
- zmm_150 is the reference operand, contains the value of 150.
- IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case $|zmm_src| < 150$, then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of $|zmm_src| \geq 150$, the selected sign of SRC1 is kept.

Thus, if zmm_src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm_src > +150, the result of VRANGE will be the maximal value of +150.
Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPS __m512_mm512_range_ps ( __m512 a, __m512 b, int imm);
VRANGEPS __m512_mm512_range_round_ps ( __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512_mm512_mask_range_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512_mm512_mask_range_round_ps ( __m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512_mm512_maskz_range_ps ( __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512_mm512_maskz_range_round_ps ( __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256_mm256_range_ps ( __m256 a, __m256 b, int imm);
VRANGEPS __m256_mm256_mask_range_ps ( __m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256_mm256_maskz_range_ps ( __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128_mm_range_ps ( __m128 a, __m128 b, int imm);
VRANGEPS __m128_mm_mask_range_ps ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128_mm_maskz_range_ps ( __mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See 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.LLG6G.0F3A.W1 51</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.
VRANGESD—Range Restriction Calculation From a pair of Scalar Float64 Values

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"
        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 bound-
ed 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 -1023while if xmm_src > +1023,
the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent
VRANGESD __m128d _mm_range_sd ( __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_range_round_sd ( __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_mask_range_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_mask_range_round_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_maskz_range_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_maskz_range_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions
Invalid, Denormal

Other Exceptions
See 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 51 /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.
VRANESS—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[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, __m128 k, __m128 a, __m128 b, int imm);
VRANGESS __m128_mm_mask_range_round_ss (__m128 s, __m128 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 VRCP14PD xmm1 [k1][z], xmm2/m128/m64bcst</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 VRCP14PD ymm1 [k1][z], ymm2/m256/m64bcst</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 VRCP14PD zmm1 [k1][z], zmm2/m512/m64bcst</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 $\infty$ 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 5-13. VRCP14PD/VRCP14SD Special Cases**

<table>
<thead>
<tr>
<th>Input value</th>
<th>Result value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \leq X \leq 2^{-1024}$</td>
<td>INF</td>
<td>Very small denormal</td>
</tr>
<tr>
<td>$-2^{-1024} \leq X \leq 0$</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 \(KL-1\)
   \(i := j \times 64\)
   IF \(k1[j] \text{ OR *no writemask* THEN}\)
      IF (EVEX.b = 1) AND (SRC *is memory*)
         THEN \(\text{DEST}[i+63:i] := \text{APPROXIMATE}(1.0/\text{SRC}[63:0]);\)
         ELSE \(\text{DEST}[i+63:i] := \text{APPROXIMATE}(1.0/\text{SRC}[i+63:i]);\)
      FI;
   ELSE
      IF *merging-masking* ; merging-masking
         THEN *\text{DEST}[i+63:i] remains unchanged*;
      ELSE ; zeroing-masking
         \(\text{DEST}[i+63:i] := 0\)
      FI;
   FI;
ENDFOR;
\(\text{DEST[MAXVL-1:VL]} := 0\)

Intel C/C++ Compiler Intrinsic Equivalent

\(\text{VRCP14PD } \_m512d \_mm512\_rcp14\_pd( \_m512d \ a);\)
\(\text{VRCP14PD } \_m512d \_mm512\_mask\_rcp14\_pd(\_m512d \ s, \_mmask8 \ k, \_m512d \ a);\)
\(\text{VRCP14PD } \_m512d \_mm512\_maskz\_rcp14\_pd(\_mmask8 \ k, \_m512d \ a);\)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 2-49, “Type E4 Class Exception Conditions”. 
VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

Opcode/Instruction | Op / En | 64/32 bit Mode Support | CPUID Feature Flag | Description
--- | --- | --- | --- | ---
EVEX.LLIG.66.0F38.W1 4D /r | A | V/V | AVX512F | Computes the approximate reciprocal of the scalar double-precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

### Instruction Operand Encoding

<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 performs a SIMD computation of the approximate reciprocal of the low double-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 (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2⁻¹⁴. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-13 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


### Operation

**VRCP14SD (EVEX version)**

IF k1[0] OR "no writemask"

THEN DEST[63:0] := APPROXIMATE(1.0/SRC2[63:0]);

ELSE

IF *merging-masking* ; merging-masking

THEN *DEST[63:0] remains unchanged*

ELSE ; zeroing-masking

DEST[63:0] := 0

FI;

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 | 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 | 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 | 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>$-2^{-128} \leq X &lt; -0$</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

A numerically exact implementation of VRCP14xx can be found at:

Operation

VRCP14PS (EVEX encoded versions)

(kL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO kL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN DEST[i+31:i] := APPROXIMATE(1.0/SRC[31:0]);
        ELSE DEST[i+31:i] := APPROXIMATE(1.0/SRC[i+31:i]);
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
            DEST[i+31:i] := 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PS __m512 _mm512_rcp14_ps( __m512 a);
VRCP14PS __m512 _mm512_mask_rcp14_ps(__m512 s, __mmask16 k, __m512 a);
VRCP14PS __m512 _mm512_maskz_rcp14_ps( __mmask16 k, __m512 a);
VRCP14PS __m256 _mm256_rcp14_ps( __m256 a);
VRCP14PS __m256 _mm256_mask_rcp14_ps(__m256 s, __mmask8 k, __m256 a);
VRCP14PS __m256 _mm256_maskz_rcp14_ps( __mmask8 k, __m256 a);
VRCP14PS __m128 _mm_rcp14_ps( __m128 a);
VRCP14PS __m128 _mm_mask_rcp14_ps(__m128 s, __mmask8 k, __m128 a);
VRCP14PS __m128 _mm_maskz_rcp14_ps( __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

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

VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values
VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LLIG.66.0F38.W0 4D /r VRCP14SS xmm1 (k1)[z], xmm2, xmm3/m32</td>
<td>A</td>
<td>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]) from xmm2 is copied to xmm1[127:32].</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 performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^-14. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-14 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/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".
VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

### 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 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:

\[
\text{dest} = \text{src} - ((\text{ROUND}(2^M \cdot \text{src})) \cdot 2^{-M});
\]

where "Round()" treats “src”, “2^M”, and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering 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.

### 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>

### Opcode/Instruction

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F3A.W1 56 / r ib</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/m64bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 56 / r ib</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</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>

### Imm8 Controls for VREDUCEPD/SD/PS/SS

- imm8[7:4]: Number of fixed points to subtract
- imm8[3]: Suppress Precision Exception
- imm8[2]: Round Select
- imm8[1:0]: Round Control Override

- imm8[7]: Suppress Precision Exception: imm8[3]=0b: Use MXCSR exception mask
- imm8[1:0]: Round Control Override
  - 0b: Round nearest even
  - 1b: Round down
  - 10b: Round up
  - 11b: Truncate

**Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS**
Handling of special case of input values are listed in Table 5-15.

| $|\text{Src1}| < 2^{-M-1}$  | Round Mode | Returned value |
|-----------------|------------|--------------|
| RNE             | $\text{Src1}$ |

| $|\text{Src1}| < 2^{-M}$  | Round Mode | Returned value |
|-----------------|------------|--------------|
| RPI, $\text{Src1} > 0$ | $\text{Round (Src1-2}^{-M})$ * |
| RPI, $\text{Src1} \leq 0$ | $\text{Src1}$ |
| RNI, $\text{Src1} \geq 0$ | $\text{Src1}$ |
| RNI, $\text{Src1} < 0$ | $\text{Round (Src1+2}^{-M})$ * |

<table>
<thead>
<tr>
<th>$\text{Src1} = \pm 0$, or Dest $= \pm 0$ (Src1!=INF)</th>
<th>Round Mode</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT RNI</td>
<td>$+0.0$</td>
<td></td>
</tr>
<tr>
<td>RNI</td>
<td>$-0.0$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{Src1} = \pm \text{INF}$</th>
<th>Round Mode</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>$+0.0$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{Src1} = \pm \text{NAN}$</th>
<th>Round Mode</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>QNaN(Src1)</td>
<td></td>
</tr>
</tbody>
</table>

* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

**Operation**

ReduceArgumentDP($\text{SRC[63:0]}, \text{imm8[7:0]}$)

```plaintext
// Check for NaN
IF (SRC[63:0] = \text{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
```
**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCEPD __m512d _mm512_mask_reduce_pd( __m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_mask_reduce_pd( __m512d s, __mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_maskz_reduce_pd( __mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m256d _mm256_mask_reduce_pd( __m256d a, int imm)
VREDUCEPD __m256d _mm256_mask_reduce_pd( __m256d s, __mmask8 k, __m256d a, int imm)
VREDUCEPD __m256d _mm256_maskz_reduce_pd( __mmask8 k, __m256d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd( __m128d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd( __m128d s, __mmask8 k, __m128d a, int imm)
VREDUCEPD __m128d _mm_maskz_reduce_pd( __mmask8 k, __m128d a, int imm)

**SIMD Floating-Point Exceptions**

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See 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:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M*\text{src}))*2^{-M};
\]

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering \(\text{src}=2^p\text{man2}\), where 'man2' is the normalized significand and 'p' is the unbiased exponent.

Then if \(\text{RC} = \text{RNE}\): \(0 \leq |\text{Reduced Result}| \leq 2^p - 1\)

Then if \(\text{RC} \neq \text{RNE}\): \(0 < |\text{Reduced Result}| < 2^p\)

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])

```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
}
```
VREDUCESD
IF k1[0] or *no writemask*
    THEN DEST[63:0] := ReduceArgumentDP(SRC2[63:0], imm8[7:0])
ELSE
    IF *merging-masking* ; merging-masking
        THEN *DEST[63:0] remains unchanged* 
    ELSE ; zeroing-masking
        THEN DEST[63:0] = 0
    FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VREDUCESD __m128d _mm_mask_reduce_sd( __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

SIMD Floating-Point Exceptions
Invalid, Precision
If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions
See 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{sae}, 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 a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[
\text{dest} = \text{src} - (\text{ROUND}(2^M\times\text{src}))\times2^{-M};
\]

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering \( \text{src} = 2^p\times\text{man2} \)

where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if \( \text{RC} = \text{RNE} \): \( 0\leq|\text{Reduced Result}|\leq2^{p-M-1} \)

Then if \( \text{RC} \neq \text{RNE} \): \( 0\leq|\text{Reduced Result}|<2^{p-M} \)

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

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>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 single-precision FP value in the low dword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

\[ \text{dest} = \text{src} - (\text{ROUND}(2^M \times \text{src})) \times 2^{-M}; \]

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\leq|\text{Reduced Result}|\leq2^{p-M-1}

Then if RC \neq RNE: 0\leq|\text{Reduced Result}|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-15.

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 \times \text{ROUND}(2^M \times \text{SRC}[31:0], \text{SPE}, \text{RC}_{\text{source}}, \text{RC}); // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[31:0] := \text{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*
        THEN "DEST[31:0] remains unchanged"
    ELSE
        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 ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Rounds packed double-precision floating point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 09 / r ib VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Rounds packed double-precision floating point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 09 / r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8</td>
<td>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

\[
\text{ROUND}(x) = 2^{-M} \times \text{Round}_\text{to}_\text{INT}(x \times 2^M, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).
VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

\[
\text{ROUND}(x) = \text{Round}_\text{to}_\text{INT}(x, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

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

<table>
<thead>
<tr>
<th>Returned value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src1=\pm \text{inf}</td>
<td>Src1</td>
</tr>
<tr>
<td>Src1=\pm \text{NAN}</td>
<td>Src1 converted to QNAN</td>
</tr>
<tr>
<td>Src1=\pm 0</td>
<td>Src1</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 _mm128_roundscale_pd(__m128d a, int imm);
VRNDSCALEPD __m128d _mm128_mask_roundscale_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm128_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".
VRNDSCALED—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits

Op / En  Opcode/ Instruction  64/32 bit Mode Support  CPUID Feature Flag  Description
A  EVEX.LLIG.66.0F3A.W1 0B /r ib  AVX512F  VRNDSCALED xmm1 {k1}{z}, xmm2, xmm3/m64{sa}, imm8  Rounds scalar double-precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register.

Instruction Operand Encoding

<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

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 VRNDSCALED is

ROUND(x) = 2^-M*Round_to_INT(x*2^M, round_ctrl),
round_ctrl = imm[3:0];
M=imm[7:4];

The operation of x*2^M is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALED is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

ROUND(x) = Round_to_INT(x, round_ctrl),
round_ctrl = 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, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_mask_roundscale_round_sd ( __m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_maskz_roundscale_sd ( __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_maskz_roundscale_round_sd ( __mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Table 2-47, “Type E3 Class Exception Conditions”.

5-552 Vol. 2C VRNDSCALESD—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits
VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits

Opcode/Instruction | Op / En | 64/32 bit Mode Support | CPUID Feature Flag | Description
--- | --- | --- | --- | ---
EVEX.128.66.0F3A.W0 08 /r ib | A | V/V | AVX512VL AVX512F | Rounds packed single-precision floating point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.

EVEX.256.66.0F3A.W0 08 /r ib | A | V/V | AVX512VL AVX512F | Rounds packed single-precision floating point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.

EVEX.512.66.0F3A.W0 08 /r ib | A | V/V | AVX512F | Rounds packed single-precision floating-point values in zmm2/m512/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask.

**Instruction Operand Encoding**

<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 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

\[
ROUND(x) = 2^{-M} \times \text{Round}_\text{to}_\text{INT}(x \times 2^M, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

\[
ROUND(x) = \text{Round}_\text{to}_\text{INT}(x, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]
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 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

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<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{sa}, 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>ModRMrr/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

\[
\text{ROUND}(x) = 2^{-M} \times \text{Round}_\text{to}_\text{INT}(x \times 2^M, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

\[
M = \text{imm}[7:4];
\]

The operation of \(x \times 2^M\) is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

\[
\text{ROUND}(x) = \text{Round}_\text{to}_\text{INT}(x, \text{round}_\text{ctrl}),
\]

\[
\text{round}_\text{ctrl} = \text{imm}[3:0];
\]

EVEX encoded version: The source operand is a XMM register or a 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[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])
}

VRNDSCALES (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

VRNDSCALES __m128 _mm_roundscale_ss ( __m128 a, __m128 b, int imm);
VRNDSCALES __m128 _mm_roundscale_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRNDSCALES __m128 __mmaskz_roundscale_round_ss ( __mmask8 k, __m128 a, __m128 b, 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 2-47, "Type E3 Class Exception Conditions".
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</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</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</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 a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an $\infty$ with the sign of the source value is returned. When the source operand is an $\pm\infty$ 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:] := APPROXIMATE(1.0/ SQRT(SRC[63:0]));
      ELSE DEST[i+63:] := APPROXIMATE(1.0/ SQRT(SRC[i+63:]));
    FI;
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:] := 0
      FI;
    FI;
  END FOR;
  DEST[MAXVL-1:VL] := 0

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

VRSQRT14PD __m512d _mm512_rsqrt14_pd( __m512d a);
VRSQRT14PD __m512d _mm512_mask_rsqrt14_pd(__m512d s, __mmask8 k, __m512d a);
VRSQRT14PD __m512d _mm512_maskz_rsqrt14_pd(__mmask8 k, __m512d a);
VRSQRT14PD __m256d _mm256_rsqrt14_pd( __m256d a);
VRSQRT14PD __m256d _mm256_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d _mm256_maskz_rsqrt14_pd(__mmask8 k, __m256d a);
VRSQRT14PD __m128d _mm_rsqrt14_pd( __m128d a);
VRSQRT14PD __m128d _mm_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d _mm_maskz_rsqrt14_pd(__mmask8 k, __m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions".

**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>2^n</td>
<td></td>
</tr>
<tr>
<td>X &lt; 0</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>X = -0</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>X = +0</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>X = +INF</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>
VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

<table>
<thead>
<tr>
<th>Opcode/Description</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.LIG.66.0F38.W1 4F /r VRSQRT14SD xmm1 [k1]{z}, xmm2, xmm3/m64</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>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
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 $+\text{ZERO}$ value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


Operation

VRSQRT14SD (EVEX version)

IF k1[0] or *no writemask*
    THEN DEST[63:0] := APPROXIMATE(1.0/ SQRT(SRC[63:0]))
ELSE
    IF *merging-masking*
        ; merging-masking
    THEN *DEST[63:0] remains unchanged*
    ELSE
        ; zeroing-masking
        THEN DEST[63:0] := 0
    FI;
FI;
DEST[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>$X = 2^{-2^n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$X = -0$</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>$X = +0$</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>$X = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

Intel C/C++ Compiler Intrinsic Equivalent

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”.
VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 4E /r</td>
<td>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 a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

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

**Operation**

**VRSQRT14PS (EVEX encoded versions)**

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

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC *is memory*)
            THEN DEST[i+31:i] := APPROXIMATE(1.0/SQRT(SRC[31:0]));
            ELSE DEST[i+31:i] := APPROXIMATE(1.0/SQRT(SRC[i+31: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_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>

### 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

Computes of the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than \(2^{-14}\). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an \(\infty\) with the sign of the source value is returned. When the source operand is an \(\infty\), zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.


### Operation

**VRSQRT14SS (EVEX version)**

\[
\text{IF } k1[0] \text{ or *no writemask*} \\
\text{THEN } \text{DEST}[31:0] := \text{APPROXIMATE}(1.0/\text{SQRT}\{\text{SRC2}[31:0]\}) \\
\text{ELSE} \\
\text{IF *merging-masking*} ; \text{merging-masking} \\
\text{THEN *DEST}[31:0] \text{ remains unchanged*} \\
\text{ELSE} ; \text{zeroing-masking} \\
\text{THEN DEST}[31:0] := 0 \\
\text{FI; } \\
\text{FI;} \\
\text{DEST}[\text{MAXVL-1:128}] := 0
\]
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>$x = 2^{-2^n}$</td>
<td>$2^n$</td>
<td></td>
</tr>
<tr>
<td>$x &lt; 0$</td>
<td>QNaN_Indefinite</td>
<td>Including -INF</td>
</tr>
<tr>
<td>$x = -0$</td>
<td>-INF</td>
<td></td>
</tr>
<tr>
<td>$x = +0$</td>
<td>+INF</td>
<td></td>
</tr>
<tr>
<td>$x = +INF$</td>
<td>+0</td>
<td></td>
</tr>
</tbody>
</table>

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

VRSQRT14SS __m128_mm_rsqrt14_ss(__m128 a, __m128 b);
VRSQRT14SS __m128_mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRSQRT14SS __m128_mm_maskz_rsqrt14_ss(__mmask8 k, __m128 a, __m128 b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Table 2-51, "Type E5 Class Exception Conditions".

---

**VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value**

**INSTRUCTION SET REFERENCE, V-Z**
VSCALEFPD—Scale Packed Float64 Values With Float64 Values

**Opcode/Instruction**

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W1 2C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Scale the packed double-precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1.</td>
</tr>
<tr>
<td>VSCALEFPD xmm1 {k1}[z], xmm2, xmm3/m128/m64bcst</td>
<td></td>
<td></td>
<td>AVX512F</td>
<td></td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 2C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL</td>
<td>Scale the packed double-precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1.</td>
</tr>
<tr>
<td>VSCALEFPD ymm1 {k1}[z], ymm2, ymm3/m256/m64bcst</td>
<td></td>
<td></td>
<td>AVX512F</td>
<td></td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 2C /r</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Scale the packed double-precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.</td>
</tr>
<tr>
<td>VSCALEFPD zmm1 {k1}[z], zmm2, zmm3/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>EVEX.vvvv (r)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

Performs a floating-point scale of the packed double-precision floating-point values in the first source operand by multiplying it by 2 power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

\[ \text{zmm1} := \text{zmm2} \times 2^\text{floor(zmm3)}. \]

Floor(zmm3) means maximum integer value \( \leq \text{zmm3} \).

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

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

Handling of special-case input values are listed in Table 5-21 and Table 5-22.

**Table 5-21. VSCALEFPD/SD/PS/SS Special Cases**

<table>
<thead>
<tr>
<th>Src1</th>
<th>±NaN</th>
<th>+Inf</th>
<th>-Inf</th>
<th>O/Denorm/Norm</th>
<th>Set IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>±SNaN</td>
<td>QNaN(Src1)</td>
<td>+INF</td>
<td>+0</td>
<td>QNaN(Src1)</td>
<td>IF either source is SNAN</td>
</tr>
<tr>
<td>±SNaN</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
<td>QNaN(Src1)</td>
<td>YES</td>
</tr>
<tr>
<td>±Inf</td>
<td>QNaN(Src2)</td>
<td>Src1</td>
<td>QNaN_Indefinite</td>
<td>Src1</td>
<td>IF Src2 is SNAN or -INF</td>
</tr>
<tr>
<td>±0</td>
<td>QNaN(Src2)</td>
<td>QNaN_Indefinite</td>
<td>Src1</td>
<td>Src1</td>
<td>IF Src2 is SNAN or +INF</td>
</tr>
<tr>
<td>Denorm/Norm</td>
<td>QNaN(Src2)</td>
<td>±INF (Src1 sign)</td>
<td>±0 (Src1 sign)</td>
<td>Compute Result</td>
<td>IF Src2 is SNAN</td>
</tr>
</tbody>
</table>
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>\geq 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 \(j := 0 \text{ TO } KL-1\)
    \(i := j * 64\)
    IF \(k1[j] \text{ OR *no writemask* THEN}\)
        IF \((EVEX.b = 1) \text{ AND (SRC2 *is memory*)}\)
            THEN DEST[\(i+63:i\)] := SCALE(SRC1[\(i+63:i\)], SRC2[63:0])
            ELSE DEST[\(i+63:i\)] := SCALE(SRC1[\(i+63:i\)], SRC2[\(i+63:i\)])
        FI;
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[\(i+63:i\)] remains unchanged*
        ELSE ; zeroing-masking
            DEST[\(i+63:i\)] := 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] := 0
**Intel C/C++ Compiler Intrinsic Equivalent**

VScaledFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int rounding);
VScaledFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int rounding);
VScaledFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int rounding);
VScaledFPD __m512d _mm512_scalef_pd(__m512d a, __m512d b);
VScaledFPD __m512d _mm512_mask_scalef_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VScaledFPD __m512d _mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);
VScaledFPD __m256d _mm256_scalef_pd(__m256d a, __m256d b);
VScaledFPD __m256d _mm256_mask_scalef_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VScaledFPD __m256d _mm256_maskz_scalef_pd(__mmask8 k, __m256d a, __m256d b);
VScaledFPD __m128d _mm_scalef_pd(__m128d a, __m128d b);
VScaledFPD __m128d _mm_mask_scalef_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VScaledFPD __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”.

INSTRUCTION SET REFERENCE, V-Z
VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

<table>
<thead>
<tr>
<th>Opcode/ Instruction</th>
<th>Op / En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.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 packed double-precision floating-point value in the first source operand by multiplying it by 2 power of the double-precision floating-point value in second source operand.

The equation of this operation is given by:

\[
\text{xmm1} := \text{xmm2} \times 2^{\text{Floor(xmm3)}}.
\]

Floor(xmm3) means maximum integer value \( \leq \text{xmm3} \).

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] * POW(2, Floor(TMP_SRC2[63:0]))
}
```
VSFSCALEFSD (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
    FI;

DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

Intel C/C++ Compiler Intrinsic Equivalent

VSFSCALEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSFSCALEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSFSCALEFSD __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”.
### VSCALEFPS—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 AVX512F</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 AVX512F</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 it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

\[ zmm1 := zmm2 \times 2^{\text{floor}(zmm3)} \]

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

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

Handling of special-case input values are listed in Table 5-21 and Table 5-23.

#### Table 5-23. Additional VSCALEFPS/SS Special Cases

<table>
<thead>
<tr>
<th>Special Case</th>
<th>Returned value</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>result</td>
<td>&lt; 2\textsuperscript{-149}</td>
</tr>
<tr>
<td></td>
<td>result</td>
<td>≥ 2\textsuperscript{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>Tuple1 Scalar</td>
<td>ModRMreg (w)</td>
<td>EVEX.vvvv (r)</td>
<td>ModRMreg/m (r)</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

 Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 power of the float32 value in second source operand.

The equation of this operation is given by:

\[ \text{xmm1} := \text{xmm2} \times 2^\text{floor(xmm3)}. \]

\( \text{Floor(xmm3)} \) means maximum integer value \( \leq \text{xmm3}. \)

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)
{
  ; Check for denormal operands
  TMP_SRC2 := SRC2
  TMP_SRC1 := SRC1
  IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
  IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
  /* SRC2 is a 32 bits floating-point value */
  DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

**VSCALEFSS (EVEX encoded version)**

IF (EVEX.b= 1) and SRC2 *is a register*
  THEN
    SET_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] := SCALE(SRC1[31:0], SRC2[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
      ELSE ; zeroing-masking
        DEST[31:0] := 0
    FI
  FI;
DEST[MAXVL-1:128] := 0

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

VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

**Other Exceptions**

See Table 2-47, "Type E3 Class Exception Conditions".
VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/E</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x (k1), xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y (k1), ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z (k1), zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x (k1), xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x (k1), ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x (k1), zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS vm64x (k1), xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y (k1), ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z (k1), zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD vm64x (k1), xmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y (k1), ymm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z (k1), zmm1</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.</td>
</tr>
</tbody>
</table>

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 (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:
- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.
• If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
• Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
• Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be 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]
  ENDIF;
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]
  ENDIF;
ENDFOR
k1[MAX_KL-1:KL] := 0

Intel C/C++ Compiler Intrinsic Equivalent
VSCATTERQPS void _mm512_i32scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERQPS void _mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPS void _mm512_i64scatter_ps(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm256_i64scatter_ps(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm256_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm_i64scatter_ps(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm_i32scatter_ps(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm_mask_i32scatter_ps(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm_i32scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm_mask_i32scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm_i64scatter_ps(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m512d 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/VSHUF132x4/VSHUF164x2—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[k1]{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 23 /r ib VSHUFF32x4 zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 23 /r ib VSHUFF64X2 ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 23 /r ib VSHUFF64X2 zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed double-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W0 43 /r ib VSHUF132x4 ymm1[k1]{z}, ymm2, ymm3/m256/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W0 43 /r ib VSHUF132x4 zmm1[k1]{z}, zmm2, zmm3/m512/m32bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.256.66.0F3A.W1 43 /r ib VSHUF164X2 ymm1[k1]{z}, ymm2, ymm3/m256/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512VL AVX512F</td>
<td>Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.</td>
</tr>
<tr>
<td>EVEX.512.66.0F3A.W1 43 /r ib VSHUF164X2 zmm1[k1]{z}, zmm2, zmm3/m512/m64bcst, imm8</td>
<td>A</td>
<td>V/V</td>
<td>AVX512F</td>
<td>Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.</td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>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 \(j := 0 TO KL-1\)
   \(i := j \times 64\)
   IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN \(\text{TMP\_SRC2}[i+63:j] := \text{SRC2}[i+63:j]\)
      ELSE \(\text{TMP\_SRC2}[i+63:j] := \text{SRC2}[i+63:j]\)
   FI;
ENDFOR;

IF \(VL = 256\)
   \(\text{TMP\_DEST}[127:0] := \text{Select2}(\text{SRC1}[255:0], \text{imm8}[0])\);
   \(\text{TMP\_DEST}[255:128] := \text{Select2}(\text{SRC2}[255:0], \text{imm8}[1])\);
ENDIF;

IF \(VL = 512\)
   \(\text{TMP\_DEST}[127:0] := \text{Select4}(\text{SRC1}[511:0], \text{imm8}[1:0])\);
   \(\text{TMP\_DEST}[255:128] := \text{Select4}(\text{SRC1}[511:0], \text{imm8}[3:2])\);
   \(\text{TMP\_DEST}[383:256] := \text{Select4}(\text{TMP\_SRC2}[511:0], \text{imm8}[5:4])\);
   \(\text{TMP\_DEST}[511:384] := \text{Select4}(\text{TMP\_SRC2}[511:0], \text{imm8}[7:6])\);
ENDIF;

FOR \(j := 0 TO KL-1\)
   \(i := j \times 64\)
   IF \(k_1[j] OR \text{*no writemask*}\)
      THEN \(\text{DEST}[i+63:j] := \text{TMP\_DEST}[i+63:j]\)
      ELSE
         IF *merging-masking*
            THEN \(\text{DEST}[i+63:j]\) remains unchanged*
         ELSE *zeroing-masking*
            THEN \(\text{DEST}[i+63:j] := 0\)
         FI
      FI
ENDIF;
ENDFOR

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

VSHUFF32x4 (EVEX 512-bit version)

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

FOR \(j := 0 TO KL-1\)
   \(i := j \times 32\)
   IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN \(\text{TMP\_SRC2}[i+31:j] := \text{SRC2}[i+31:j]\)
      ELSE \(\text{TMP\_SRC2}[i+31:j] := \text{SRC2}[i+31:j]\)
   FI;
ENDFOR;

IF \(VL = 256\)
   \(\text{TMP\_DEST}[127:0] := \text{Select2}(\text{SRC1}[255:0], \text{imm8}[0])\);
   \(\text{TMP\_DEST}[255:128] := \text{Select2}(\text{SRC2}[255:0], \text{imm8}[1])\);
ENDIF;

IF \(VL = 512\)
   \(\text{TMP\_DEST}[127:0] := \text{Select4}(\text{SRC1}[511:0], \text{imm8}[1:0])\);
   \(\text{TMP\_DEST}[255:128] := \text{Select4}(\text{SRC1}[511:0], \text{imm8}[3:2])\);
   \(\text{TMP\_DEST}[383:256] := \text{Select4}(\text{TMP\_SRC2}[511:0], \text{imm8}[5:4])\);
   \(\text{TMP\_DEST}[511:384] := \text{Select4}(\text{TMP\_SRC2}[511:0], \text{imm8}[7:6])\);
ENDIF;

FOR \(j := 0 TO KL-1\)
   \(i := j \times 32\)
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[255:128] := Select4(SRC1[511:0], imm8[3:2])
  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
INSTRUCTION SET REFERENCE, V-Z

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFF32x4 __m512i _mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFF32x4 __m512i _mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFF32x4 __m512i _mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFF32x4 __m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFF32x4 __m256i _mm256_mask_shuffle_i32x4(__m256i s, __mmask16 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m256i _mm256_maskz_shuffle_i32x4(__mmask16 k, __m256i a, __m256i b, int imm);

VSHUFF32x4 __m512d _mm512_shuffle_f32x4(__m512d a, __m512d b, int imm);
VSHUFF32x4 __m512d _mm512_mask_shuffle_f32x4(__m512d s, __mmask16 k, __m512d a, __m512d b, int imm);
VSHUFF32x4 __m512d _mm512_maskz_shuffle_f32x4(__mmask16 k, __m512d a, __m512d 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.
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</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.</td>
</tr>
<tr>
<td>VTESTPS xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0E /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.</td>
</tr>
<tr>
<td>VTESTPS ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.128.66.0F38.W0 0F /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.</td>
</tr>
<tr>
<td>VTESTPD xmm1, xmm2/m128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VEX.256.66.0F38.W0 0F /r</td>
<td>RM</td>
<td>V/V</td>
<td>AVX</td>
<td>Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.</td>
</tr>
<tr>
<td>VTESTPD ymm1, ymm2/m256</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

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

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)**


IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
   THEN ZF := 1;
   ELSE ZF := 0;


IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
   THEN CF := 1;
   ELSE CF := 0;

DEST (unmodified)

AF := OF := PF := SF := 0;

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

TEMP[255:0] := SRC[255:0] AND DEST[255:0]

   THEN ZF := 1;
   ELSE ZF := 0;

TEMP[255:0] := SRC[255:0] AND NOT DEST[255:0]

   THEN CF := 1;
   ELSE CF := 0;

DEST (unmodified)

AF := OF := PF := SF := 0;

**VTESTPD (128-bit version)**


IF (TEMP[63] = TEMP[127] = 0)
   THEN ZF := 1;
   ELSE ZF := 0;


IF (TEMP[63] = TEMP[127] = 0)
   THEN CF := 1;
   ELSE CF := 0;

DEST (unmodified)

AF := OF := PF := SF := 0;

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

TEMP[255:0] := SRC[255:0] AND DEST[255:0]

   THEN ZF := 1;
   ELSE ZF := 0;

TEMP[255:0] := SRC[255:0] AND NOT DEST[255:0]

   THEN CF := 1;
   ELSE CF := 0;

DEST (unmodified)

AF := OF := PF := SF := 0;
Intel C/C++ Compiler Intrinsic Equivalent

**VTESTPS**

```c
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**

```c
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

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
\( \text{simd_reg_file}[i] \) 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: \_mm256_zeroall()

SIMD Floating-Point Exceptions
None.

Other Exceptions
See Table 2-25, “Type 8 Class Exception Conditions”.

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>

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEX.256.0F.WIG 77</td>
<td>ZO</td>
<td>V/V</td>
<td>AVX</td>
<td>Zero some of the XMM, YMM and ZMM registers.</td>
</tr>
<tr>
<td>VZEROALL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 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</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>

**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**

```plaintext
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

| Opcode | Instruction | Op/En | 64-Bit Mode | Compat/ 
Leg Mode | Description |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>

**Description**

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1, for more information on using the WAIT/FWAIT instruction.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

CheckForPendingUnmaskedFloatingPointExceptions;

**FPU Flags Affected**

The C0, C1, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.
WBINVD—Write Back and Invalidate Cache

<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 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 WBNOINVD</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>
</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</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>WRFSBASE r32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 REX.W OF AE /2</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>WRFSBASE r64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 0F AE /3</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>WRGSBASE r32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3 REX.W OF AE /3</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>
<tr>
<td>WRGSBASE r64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>Compat/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

\[\text{MSR}[\text{ECX}] := \text{EDX:EAX};\]

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 WRPKRU</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 WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX and RAX are ignored.

WRPKRU will never execute speculatively. Memory accesses affected by PKRU register will not execute (even speculatively) until all prior executions of WRPKRU 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

WRPKRU: void _wrpkru(uint32_t);

Protected Mode Exceptions

#GP(0)  IF ECX ≠ 0.
        IF EDX ≠ 0.
#UD     IF the LOCK prefix is used.
        IF CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.
64-Bit Mode Exceptions
Same exceptions as in protected mode.
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</td>
<td>MR</td>
<td>V/V</td>
<td>CET_SS</td>
<td>Write 4 bytes to shadow stack.</td>
</tr>
<tr>
<td>WRSSD</td>
<td>m32, r32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REX.W OF 38 F6</td>
<td>MR</td>
<td>V/N.E.</td>
<td>CET_SS</td>
<td>Write 8 bytes to shadow stack.</td>
</tr>
<tr>
<td>WRSSQ</td>
<td>m64, r64</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>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;
    ELSE
      DEST_LA = Linear_Address(mem operand)
      IF (operand size is 64 bit)
        THEN
          (* Destination not 8B aligned *)
          IF DEST_LA[2:0] = 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] = 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 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 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 CPL < 3.
  Other terminal and non-terminal faults.
WRUSSD/WRUSSQ—Write to User Shadow Stack

**Opcode/Instruction**  | **Op/En** | **64/32 bit Mode Support** | **CPUID Feature Flag** | **Description**  
---|---|---|---|---  
66 0F 38 F5 l(11):rrr:bbb  | MR | V/V | CET_SS | Write 4 bytes to shadow stack.  
WRUSSD m32, r32  
66 REX.W 0F 38 F5 l(11):rrr:bbb  | MR | V/N.E. | CET_SS | Write 8 bytes to shadow stack.  
WRUSSQ m64, r64

**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 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;  
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.
XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>64/32bit Mode Support</th>
<th>CPUID Feature Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2 XACQUIRE</td>
<td>V/V</td>
<td>HLE¹</td>
<td>A hint used with an “XACQUIRE-enabled” instruction to start lock elision on the instruction memory operand address.</td>
</tr>
<tr>
<td>F3 XRELEASE</td>
<td>V/V</td>
<td>HLE</td>
<td>A hint used with an “XRELEASE-enabled” instruction to end lock elision on the instruction memory operand address.</td>
</tr>
</tbody>
</table>

NOTES:
1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The “MOV mem, reg” (Opcode 88H/89H) and “MOV mem, imm” (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume I*, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.
Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode. For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to:

- REPNE/REPNZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to:

- REP/REPE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

Operation

XACQUIRE
IF XACQUIRE-enabled instruction
THEN
  IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN
    HLE_NEST_COUNT++
    IF (HLE_NEST_COUNT = 1) THEN
      HLE_ACTIVE := 1
      IF 64-bit mode THEN
        restartRIP := instruction pointer of the XACQUIRE-enabled instruction
      ELSE
        restartEIP := instruction pointer of the XACQUIRE-enabled instruction
      FI;
      Enter HLE Execution (* record register state, start tracking memory state *)
    FI; (* HLE_NEST_COUNT = 1 *)
    IF ElisionBufferAvailable THEN
      Allocate elision buffer
      Record address and data for forwarding and commit checking
      Perform elision
    ELSE
      Perform lock acquire operation transactionally but without elision
    FI;
  ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)
    GOTO HLE_ABORT_PROCESSING
  FI;
ELSE (* Treat instruction as non-XACQUIRE F2H prefixed legacy instruction *)
FI;
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;
        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;
                    ELSE
                        GOTO HLE_ABORT_PROCESSING
                    FI;
                FI; (* HLE_NEST_COUNT > 0 *)
            ELSE
                Treat instruction as non-XRELEASE F3H prefixed legacy instruction
            FI;
        FI; (* For any HLE abort condition encountered during HLE execution *)

HLE_ABORT_PROCESSING:
    HLE_ACTIVE := 0
    HLE_NEST_COUNT := 0
    Restore architectural register state
    Discard memory updates performed in transaction
    Free any allocated lock elision buffers
    IF 64-bit mode
        THEN
            RIP := restartRIP
        ELSE
            EIP := restartEIP
        FI;
    Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END
SIMD Floating-Point Exceptions
None

Other Exceptions
#GP(0) If the use of prefix causes instruction length to exceed 15 bytes.
XABORT — Transactional Abort

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C6 F8 ib XABORT imm8</td>
</tr>
</tbody>
</table>

**Description**

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

**Operation**

XABORT

IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
    FI;

(* For any RTM abort condition encountered during RTM execution *)

RTM_ABORT_PROCESSING:
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT:= 0;
    RTM_ACTIVE:= 0;
    IF 64-bit Mode
        THEN
            RIP:= fallbackRIP;
        ELSE
            EIP := fallbackEIP;
        FI;
    END

**Flags Affected**

None

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

XABORT: 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**

**Description**

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

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

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

**IA-32 Architecture Compatibility**

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

**Operation**

\[
\text{TEMP} := \text{SRC} + \text{DEST}; \\
\text{SRC} := \text{DEST}; \\
\text{DEST} := \text{TEMP};
\]

**Flags Affected**

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

**Protected Mode Exceptions**

* #GP(0) If the destination is located in a non-writable segment.
  * 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

**XBEGIN**

```plaintext
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;
                ELSE
                    RTM_ACTIVE := 1
                    Enter RTM Execution (* record register state, start tracking memory state*)
```
FI; (* RTM_NEST_COUNT = 1 *)  
ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)  
    GOTO RTM_ABORT_PROCESSING  
FI;  
(* For any RTM abort condition encountered during RTM execution *)  
RTM_ABORT_PROCESSING:  
    Restore architectural register state  
    Discard memory updates performed in transaction  
    Update EAX with status  
    RTM_NEST_COUNT := 0  
    RTM_ACTIVE := 0  
    IF 64-bit mode  
        THEN  
            RIP := fallbackRIP  
        ELSE  
            EIP := fallbackEIP  
    FI;  
END  

Flags Affected  
None  

Intel C/C++ Compiler Intrinsic Equivalent  
XBEGIN: unsigned int _xbegin( void );  

SIMD Floating-Point Exceptions  
None  

Protected Mode Exceptions  
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.  
    If LOCK prefix is used.  
#GP(0) If the fallback address is outside the CS segment.  

Real-Address Mode Exceptions  
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.  
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.  
    If LOCK prefix is used.  

Virtual-8086 Mode Exceptions  
#GP(0) If the fallback address is outside the address space 0000H and FFFFH.  
#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.  
    If LOCK prefix is used.  

Compatibility Mode Exceptions  
Same exceptions as in protected mode.
**64-bit Mode Exceptions**

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.
   If LOCK prefix is used.

#GP(0)	If the fallback address is non-canonical.
XCHG—Exchange Register/Memory with Register

### Opcode

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>90+rw</td>
<td>XCHG AX, r16</td>
<td>0</td>
<td>Valid</td>
<td>Valid</td>
<td>Exchange r16 with AX.</td>
</tr>
<tr>
<td>90+rw</td>
<td>XCHG AX16, 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 r8 (byte register) with byte from r/m8.</td>
</tr>
<tr>
<td>REX + 86 /r</td>
<td>XCHG r/m8*, r8*</td>
<td>MR</td>
<td>Valid</td>
<td>N.E.</td>
<td>Exchange r8 (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 doubleword from r/m32 with r32.</td>
</tr>
<tr>
<td>REX.W + 87 /r</td>
<td>XCHG r64, r/m64</td>
<td>RM</td>
<td>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:r/m (r, w)</td>
<td>ModRM:reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM:reg (w)</td>
<td>ModRM:r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 8 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

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

XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

Operation
TEMP := DEST;
DEST := SRC;
SRC := TEMP;

Flags Affected
None.

Protected Mode Exceptions
#GP(0) If either operand is in a non-writable segment.
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>Operand2</th>
<th>Operand3</th>
<th>Operand4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

**Operation**

XEND

IF (RTM_ACTIVE = 0) THEN
  SIGNAL #GP
ELSE
  RTM_NEST_COUNT--
  IF (RTM_NEST_COUNT = 0) THEN
    Try to commit transaction
    IF fail to commit transactional execution
      THEN
        GOTO RTM_ABORT_PROCESSING;
      ELSE (* commit success *)
        RTM_ACTIVE := 0
      FI;
    FI;
  FI;
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

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.
**XLAT/XLATB—Table Look-up Translation**

**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-operands” form and the “no-operands” form. The explicit-operands form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a "short form" of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

IF AddressSize = 16

THEN

AL := (DS:BX + ZeroExtend(AL));
ELSE IF (AddressSize = 32)

AL := (DS:EBX + ZeroExtend(AL)); Fl;
ELSE (AddressSize = 64)

AL := (RBX + ZeroExtend(AL));
Fl;

**Flags Affected**
None.

**Protected Mode Exceptions**

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

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

#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

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

**Virtual-8086 Mode Exceptions**

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0) If a memory operand effective address is outside the SS segment limit.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.
#GP(0) If the memory address is in a non-canonical form.
#PF(fault-code) If a page fault occurs.
#UD If the LOCK prefix is used.
XOR—Logical Exclusive OR

**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.

### Opcode Instruction

<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>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AL/AX/EAX/RAX</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MI</td>
<td>ModRM/r/m (r, w)</td>
<td>imm8/16/32</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>MR</td>
<td>ModRM/r/m (r, w)</td>
<td>ModRM/reg (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>RM</td>
<td>ModRM/reg (r, w)</td>
<td>ModRM/r/m (r)</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Notes:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.
In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**Operation**

DEST := DEST XOR SRC;

**Flags Affected**

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

**Protected Mode Exceptions**

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

**Real-Address Mode Exceptions**

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

**Virtual-8086 Mode Exceptions**

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

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

- **#SS(0)** If a memory address referencing the SS segment is in a non-canonical form.
- **#GP(0)** If the memory address is in a non-canonical form.
- **#PF(fault-code)** If a page fault occurs.
- **#AC(0)** If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- **#UD** If the LOCK prefix is used but the destination is not a memory operand.
### XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

<table>
<thead>
<tr>
<th>Opcode/Instruction</th>
<th>Op/En</th>
<th>64/32 bit Mode Support</th>
<th>CPUID Feature Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 \( KL-1 \)

\[
i := j \times 64
\]

If \( k1[\square] \) OR *no writemask* THEN

- If \((EVEX.b == 1) AND (SRC2 *is memory*)\)
  - THEN \( \text{DEST}[i+63:i] := \text{SRC1}[i+63:i] \text{BITWISE XOR SRC2}[63:0] \)
  - ELSE \( \text{DEST}[i+63:i] := \text{SRC1}[i+63:i] \text{BITWISE XOR SRC2}[i+63:i] \)

- ELSE

  - IF *merging-masking* (merging-masking)
    - THEN \( *\text{DEST}[i+63:i] \text{remains unchanged}*\)
    - ELSE *zeroing-masking* (zeroing-masking)
      - \( \text{DEST}[i+63:i] = 0 \)

- ENDIF

ENDIF

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

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

\( \text{DEST}[63:0] := \text{SRC1}[63:0] \text{BITWISE XOR SRC2}[63:0] \)

\( \text{DEST}[127:64] := \text{SRC1}[127:64] \text{BITWISE XOR SRC2}[127:64] \)


\( \text{DEST}[\text{MAXVL}-1:256] := 0 \)

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

\( \text{DEST}[63:0] := \text{SRC1}[63:0] \text{BITWISE XOR SRC2}[63:0] \)

\( \text{DEST}[127:64] := \text{SRC1}[127:64] \text{BITWISE XOR SRC2}[127:64] \)

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

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

\( \text{DEST}[63:0] := \text{DEST}[63:0] \text{BITWISE XOR SRC}[63:0] \)

\( \text{DEST}[127:64] := \text{DEST}[127:64] \text{BITWISE XOR SRC}[127:64] \)

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

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

- \( \text{VXORPD}\_\text{m512d\_mm512\_xor\_pd}(\_\text{m512d\_a, m512d\_b}) \)
- \( \text{VXORPD}\_\text{m512d\_mm512\_mask\_xor\_pd}(\_\text{m512d\_a, mmask8\_m, m512d\_b}) \)
- \( \text{VXORPD}\_\text{m512d\_mm512\_maskz\_xor\_pd}(\_\text{mmask8\_m, m512d\_a}) \)
- \( \text{VXORPD}\_\text{m256d\_mm256\_xor\_pd}(\_\text{m256d\_a, m256d\_b}) \)
- \( \text{VXORPD}\_\text{m256d\_mm256\_mask\_xor\_pd}(\_\text{m256d\_a, mmask8\_m, m256d\_b}) \)
- \( \text{VXORPD}\_\text{m256d\_mm256\_maskz\_xor\_pd}(\_\text{mmask8\_m, m256d\_a}) \)
- \( \text{XORPD}\_\text{m128d\_mm128\_xor\_pd}(\_\text{m128d\_a, m128d\_b}) \)
- \( \text{VXORPD}\_\text{m128d\_mm128\_mask\_xor\_pd}(\_\text{m128d\_a, mmask8\_m, m128d\_b}) \)
- \( \text{VXORPD}\_\text{m128d\_mm128\_maskz\_xor\_pd}(\_\text{mmask8\_m, m128d\_a}) \)

**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**

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

**EVEX.512 encoded version:** The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

**VEX.256 and EVEX.256 encoded versions:** The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

**VEX.128 and EVEX.128 encoded versions:** The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

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

VXORPS (EVEX encoded versions)

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

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

\(i := j \times 32\)

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

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

\(\text{DEST}[i+31:i] := \text{SRC1}[i+31:i] \text{ BITWISE XOR SRC2}[31:0];\)

\(\text{ELSE DEST}[i+31:i] := \text{SRC1}[i+31:i] \text{ BITWISE XOR SRC2}[i+31:i];\)

\(\text{FI;}\)

\(\text{ELSE}\)

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

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

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

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

\(\text{FI}\)

\(\text{FI;}\)

\(\text{ENDFOR}\)

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

VXORPS (VEX.256 encoded version)

\(\text{DEST}[31:0] := \text{SRC1}[31:0] \text{ BITWISE XOR SRC2}[31:0]\)

\(\text{DEST}[63:32] := \text{SRC1}[63:32] \text{ BITWISE XOR SRC2}[63:32]\)

\(\text{DEST}[95:64] := \text{SRC1}[95:64] \text{ BITWISE XOR SRC2}[95:64]\)

\(\text{DEST}[127:96] := \text{SRC1}[127:96] \text{ BITWISE XOR SRC2}[127:96]\)

\(\text{DEST}[159:128] := \text{SRC1}[159:128] \text{ BITWISE XOR SRC2}[159:128]\)

\(\text{DEST}[191:160] := \text{SRC1}[191:160] \text{ BITWISE XOR SRC2}[191:160]\)

\(\text{DEST}[223:192] := \text{SRC1}[223:192] \text{ BITWISE XOR SRC2}[223:192]\)

\(\text{DEST}[255:224] := \text{SRC1}[255:224] \text{ BITWISE XOR SRC2}[255:224].\)

\(\text{DEST[MAXVL-1:256]} := 0\)

VXORPS (VEX.128 encoded version)

\(\text{DEST}[31:0] := \text{SRC1}[31:0] \text{ BITWISE XOR SRC2}[31:0]\)

\(\text{DEST}[63:32] := \text{SRC1}[63:32] \text{ BITWISE XOR SRC2}[63:32]\)

\(\text{DEST}[95:64] := \text{SRC1}[95:64] \text{ BITWISE XOR SRC2}[95:64]\)

\(\text{DEST}[127:96] := \text{SRC1}[127:96] \text{ BITWISE XOR SRC2}[127:96]\)

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

XORPS (128-bit Legacy SSE version)

\(\text{DEST}[31:0] := \text{SRC1}[31:0] \text{ BITWISE XOR SRC2}[31:0]\)

\(\text{DEST}[63:32] := \text{SRC1}[63:32] \text{ BITWISE XOR SRC2}[63:32]\)

\(\text{DEST}[95:64] := \text{SRC1}[95:64] \text{ BITWISE XOR SRC2}[95:64]\)

\(\text{DEST}[127:96] := \text{SRC1}[127:96] \text{ BITWISE XOR SRC2}[127:96]\)

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

Intel C/C++ Compiler Intrinsic Equivalent

\(\text{VXORPS } \_\_m512 \_\_mm512\_xor\_ps (\_m512 a, \_m512 b);\)

\(\text{VXORPS } \_\_m512 \_\_mm512\_mask\_xor\_ps (\_m512 a, \_m512 b);\)

\(\text{VXORPS } \_\_m512 \_\_mm512\_maskz\_xor\_ps (\_m512 a, \_m512 b);\)

\(\text{VXORPS } \_\_m256 \_\_mm256\_xor\_ps (\_m256 a, \_m256 b);\)

\(\text{VXORPS } \_\_m256 \_\_mm256\_mask\_xor\_ps (\_m256 a, \_m256 b);\)

\(\text{VXORPS } \_\_m256 \_\_mm256\_maskz\_xor\_ps (\_m256 a, \_m256 b);\)

\(\text{VXORPS } \_\_m128 \_\_mm128\_xor\_ps (\_m128 a, \_m128 b);\)

\(\text{VXORPS } \_\_m128 \_\_mm128\_mask\_xor\_ps (\_m128 a, \_m128 b);\)

\(\text{VXORPS } \_\_m128 \_\_mm128\_maskz\_xor\_ps (\_m128 a, \_m128 b);\)
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>ModRMrr/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[i] = 0, XRSTOR does not update state component i.\(^1\)

- If RFBM[i] = 1 and bit i is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component i.

- If RFBM[i] = 1 and XSTATE_BV[i] = 1, XRSTOR loads state component i from the XSAVE area.

- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.

- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.

- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component i for which RFBM[i] = 1 and XSTATE_BV[i] = 1; it tracks as modified any state component i for which RFBM[i] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

---

\(^1\) 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;
FORMAT = COMPMAK AND 7FFFFFFF_FFFFFFFH;
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
    XMUNUSE[0] := 1;
    load x87 state from legacy region of XSAVE area;
ELSIF TO_BE_INITIALIZED[0] = 1
  THEN
    XMUNUSE[0] := 0;
    initialize x87 state;
FI;

IF TO_BE_RESTORED[1] = 1
  THEN
    XMUNUSE[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;
    XMUNUSE[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
          XMUNUSE[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
        XMUNUSE[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 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).

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 mem</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 XRSTORS64 mem</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>

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>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, COMPRESS⟩;

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 (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 XCRO.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. 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 RFBM[i] = 1.1

• 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 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 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**

\[
\text{RFBM} := \text{XCR0 AND EDX:EAX}; \quad \text{/* bitwise logical AND */}
\]

\[
\text{OLD_BV} := \text{XSTATE_BV field from XSAVE header};
\]

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;

\[
\text{XSTATE_BV field in XSAVE header} := \text{(OLD_BV AND NOT RFBM)} \text{ OR (XINUSE AND RFBM)};
\]

**Flags Affected**

None.

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

XSAVE: \text{void } _xsave( \text{void *}, \text{unsigned } \text{__int64});

XSAVE: \text{void } _xsave64( \text{void *}, \text{unsigned } \text{__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).
**XSAVEC—Save Processor Extended States with Compaction**

### 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>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 \( \text{XINUSE}[i] = 1 \). (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. (See Section 13.4.2, "XSAVE Header" of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1.) XSAVEC sets bits 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**

\[ \begin{align*}
RFBM & := XCR0 \text{ AND } EDX:EAX; \quad /* \text{bitwise logical AND} */ \\
 TO\_BE\_SAVED & := RFBM \text{ AND } XINUSE; \quad /* \text{bitwise logical AND} */ \\
\text{If } MXCSR \neq 1F80H \text{ AND } RFBM[1] \\
\text{TO\_BE\_SAVED}[1] & = 1; \\
F; \\
\text{IF } TO\_BE\_SAVED[0] = 1 \\
\text{THEN store x87 state into legacy region of XSAVE area;} \\
F; \\
\text{IF } TO\_BE\_SAVED[1] = 1 \\
\text{THEN store SSE state into legacy region of XSAVE area;} \quad // \text{this step saves the XMM registers, MXCSR, and MXCSR\_MASK} \\
F; \\
\text{NEXT\_FEATURE\_OFFSET} & = 576; \quad // \text{Legacy area and XSAVE header consume 576 bytes} \\
\text{FOR } i := 2 \text{ TO } 62 \\
\text{IF } RFBM[i] = 1 \\
\text{THEN} \\
\text{IF } TO\_BE\_SAVED[i] \\
\text{THEN save XSAVE state component } i \text{ at offset NEXT\_FEATURE\_OFFSET from base of XSAVE area;} \\
F; \\
\text{NEXT\_FEATURE\_OFFSET} & = \text{NEXT\_FEATURE\_OFFSET} + n (n \text{ enumerated by CPUID(EAX=0DH,ECX=i):EAX}); \\
F; \\
\text{ENDFOR;} \\
\text{XSTATE\_BV field in XSAVE header} & := \text{TO\_BE\_SAVED}; \\
\text{XCOMP\_BV field in XSAVE header} & := RFBM \text{ OR } 80000000_00000000H;
\end{align*} \]

**Flags Affected**

None.

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

\[ \begin{align*}
\text{XSAVEC: } & \text{ void } _\text{xsavec}(\text{ void } *, \text{ unsigned int64);} \\
\text{XSAVEC64: } & \text{ void } _\text{xsavec64}(\text{ void } *, \text{ unsigned int64);}
\end{align*} \]

**Protected Mode Exceptions**

\[ \begin{align*}
\#GP(0) & \quad \text{If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.} \\
\text{If a memory operand is not aligned on a 64-byte boundary, regardless of segment.} \\
\#SS(0) & \quad \text{If a memory operand effective address is outside the SS segment limit.} \\
\#PF(fault-code) & \quad \text{If a page fault occurs.} \\
\#NM & \quad \text{If CR0.TS[bit 3] = 1.} \\
\#UD & \quad \text{If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.} \\
\text{If CR4.OSXSAVE[bit 18] = 0.} \\
\text{If the LOCK prefix is used.}
\end{align*} \]
#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 (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 XCRO.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. 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 XSAVE 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**

\[
\begin{align*}
RFBM & := XCR0 \text{ AND EDX:EAX}; \quad /* \text{bitwise logical AND} */ \\
OLD_BV & := \text{XSTATE_BV field from XSAVE header}; \\
TO\_BE\_SAVED & := RFBM \text{ AND XINUSE}; \\
\text{IF in VMX non-root operation} & \\
\quad \text{THEN VMXNR} := 1; \\
\quad \text{ELSE VMXNR} := 0; \\
\text{FI;} \\
\text{LAXA} & := \text{linear address of XSAVE area}; \\
\text{IF XRSTOR}\_\text{INFO} = \langle \text{CPL,VMXNR,LAXA,00000000_00000000H} \rangle & \\
\quad \text{THEN TO\_BE\_SAVED} := \text{TO\_BE\_SAVED AND XMODIFIED}; \\
\text{FI;} \\
\text{IF TO\_BE\_SAVED}[0] = 1 & \\
\quad \text{THEN store x87 state into legacy region of XSAVE area;} \\
\text{FI;} \\
\text{IF TO\_BE\_SAVED}[1] & \\
\quad \text{THEN store XMM registers into legacy region of XSAVE area;} \quad // \text{this step does not save MXCSR or MXCSR\_MASK} \\
\text{FI;} \\
\text{IF RFBM}[1] = 1 \text{ or RFBM}[2] = 1 & \\
\quad \text{THEN store MXCSR and MXCSR\_MASK into legacy region of XSAVE area;} \\
\text{FI;} \\
\text{FOR } i := 2 \text{ TO 62} & \\
\quad \text{IF TO\_BE\_SAVED}[i] = 1 \\
\quad \quad \text{THEN save XSAVE state component } i \text{ at offset } n \text{ from base of XSAVE area} (n \text{ enumerated by CPUID(EAX=0DH,ECX}=i):\text{EBX}); \\
\quad \text{FI;} \\
\text{ENDFOR;} \\
\text{XSTATE_BV field in XSAVE header} & := (OLD\_BV \text{ AND NOT RFBM}) \text{ OR (XINUSE AND RFBM);} \\
\end{align*}
\]

**Flags Affected**

None.

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

\[
\begin{align*}
\text{XSAVEOPT:} & \quad \text{void } _{\text{xsaveopt}}(\text{void } *, \text{unsigned } \_\text{int64}); \\
\text{XSAVEOPT:} & \quad \text{void } _{\text{xsaveopt64}}(\text{void } *, \text{unsigned } \_\text{int64}); \\
\end{align*}
\]

**Protected Mode Exceptions**

\[
\begin{align*}
\#GP(0) & \quad \text{If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.} \\
\quad & \quad \text{If a memory operand is not aligned on a 64-byte boundary, regardless of segment.} \\
\#SS(0) & \quad \text{If a memory operand effective address is outside the SS segment limit.} \\
\#PF(\text{fault-code}) & \quad \text{If a page fault occurs.} \\
\#NM & \quad \text{If CR0.TS[bit 3] = 1.} \\
\end{align*}
\]
#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

### 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>NP 0F C7 /5 XSAVES</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</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>

### Instruction Operand Encoding

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

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1. 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 XSAVE. XSAVES differs from XSAVE in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component i only if RFBM[i] = 1 and XINUSE[i] = 1.1 (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State” 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**

\[
\text{RFBM} := (\text{XCR0 OR IA32_XSS}) \text{ AND EDX:EAX};  \quad /* \text{bitwise logical OR and AND} */
\]

IF in VMX non-root operation
  THEN \( \text{VMXNR} := 1; \)
  ELSE \( \text{VMXNR} := 0; \)
FI;

\( \text{LAXA} := \text{linear address of XSAVE area}; \)

\( \text{COMPMASK} := \text{RFBM OR } 80000000_00000000\text{H}; \)

\( \text{TO\_BE\_SAVED} := \text{RFBM AND XINUSE}; \)

IF XRSTOR_INFO = \( \langle \text{CPL, VMXNR, LAXA, COMPMASK} \rangle \)
  THEN \( \text{TO\_BE\_SAVED} := \text{TO\_BE\_SAVED AND XMODIFIED}; \)
FI;

IF \( \text{MXCSR} \neq 1F80\text{H AND RFBM}[1] \)
  THEN \( \text{TO\_BE\_SAVED}[1] = 1; \)
FI;

IF \( \text{TO\_BE\_SAVED}[0] = 1 \)
  THEN store x87 state into legacy region of XSAVE area;
FI;

IF \( \text{TO\_BE\_SAVED}[1] = 1 \)
  THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR\_MASK
FI;

\( \text{NEXT\_FEATURE\_OFFSET} = 576; \quad /* \text{Legacy area and XSAVE header consume 576 bytes} */
\]

FOR \( i := 2 \) TO \( 62 \)
  IF \( \text{RFBM}[i] = 1 \)
    THEN
      IF \( \text{TO\_BE\_SAVED}[i] \)
        THEN
          save XSAVE state component \( i \) at offset \( \text{NEXT\_FEATURE\_OFFSET} \) from base of XSAVE area;
          IF \( i = 8 \) \quad // state component 8 is for PT state
            THEN \( \text{IA32\_RTIT\_CTL.TraceEn}[\text{bit 0}] := 0; \)
          FI;
          FI;
          \( \text{NEXT\_FEATURE\_OFFSET} = \text{NEXT\_FEATURE\_OFFSET} + n \) \( (n \text{ enumerated by CPUID(EAX=0DH,ECX=i):EAX}); \)
        FI;
    FI;
ENDFOR;

\( \text{NEW\_HEADER} := \text{RFBM AND XINUSE}; \)

IF \( \text{MXCSR} \neq 1F80\text{H AND RFBM}[1] \)
  THEN \( \text{NEW\_HEADER}[1] = 1; \)
FI;

\( \text{XSTATE\_BV field in XSAVE header} := \text{NEW\_HEADER}; \)
\( \text{XCOMP\_BV field in XSAVE header} := \text{COMPMASK}; \)

**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
\[ \text{XSETBV: } \text{void}_xsetbv(\text{unsigned int, unsigned __int64}); \]

### Protected Mode Exceptions
- **#GP(0)**
  - If the current privilege level is not 0.
  - If an invalid XCR is specified in ECX.
  - If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
  - If an attempt is made to clear bit 0 of XCR0.
  - If an attempt is made to set XCR0[2:1] to 10b.
- **#UD**
  - If CPUID.01H:ECX.XSAVE[bit 26] = 0.
  - If CR4.OSXSAVE[bit 18] = 0.
  - If the LOCK prefix is used.
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>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>Operand 1</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.