



# Intel® 64 and IA-32 Architectures Software Developer's Manual

## Volume 2C: Instruction Set Reference, V

**NOTE:** The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of ten volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference, A-L*, Order Number 253666; *Instruction Set Reference, M-U*, Order Number 253667; *Instruction Set Reference, V*, Order Number 326018; *Instruction Set Reference, W-Z*, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831; *Model-Specific Registers*, Order Number 335592. Refer to all ten volumes when evaluating your design needs.

Order Number: 326018-082US  
December 2023

## **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. Lower 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations**

| Imm   | [7:4]      |                |                |                |                |                |                 |                 |
|-------|------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|
| [3:0] | 0H         | 1H             | 2H             | 3H             | 4H             | 5H             | 6H              | 7H              |
| 00H   | FALSE      | andAnorBC      | norBnandAC     | andA!B         | norCnandBA     | andA!C         | andAxorBC       | andAnandBC      |
| 01H   | norABC     | norCB          | norBxorAC      | A?!B:norBC     | norCxorBA      | A?!C:norBC     | A?xorBC:norBC   | A?nandBC:norBC  |
| 02H   | andCnorBA  | norBxnorAC     | andC!B         | norBnorAC      | C?norBA:andBA  | C?norBA:A      | C?!B:andBA      | C?!B:A          |
| 03H   | norBA      | norBandAC      | C?!B:norBA     | !B             | C?norBA:xnorBA | A?!C:!B        | A?xorBC:!B      | A?nandBC:!B     |
| 04H   | andBnorAC  | norCxnorBA     | B?norAC:andAC  | B?norAC:A      | andB!C         | norCnorBA      | B?!C:andAC      | B?!C:A          |
| 05H   | norCA      | norCandBA      | B?norAC:xnorAC | A?!B:!C        | B?!C:norAC     | !C             | A?xorBC:!C      | A?nandBC:!C     |
| 06H   | norAxnorBC | A?norBC:xorBC  | B?norAC:C      | xorBorAC       | C?norBA:B      | xorCorBA       | xorCB           | B?!C:orAC       |
| 07H   | norAandBC  | minorABC       | C?!B:!A        | nandBorAC      | B?!C:!A        | nandCorBA      | A?xorBC:nandBC  | nandCB          |
| 08H   | norAnandBC | A?norBC:andBC  | andCxorBA      | A?!B:andBC     | andBxorAC      | A?!C:andBC     | A?xorBC:andBC   | xorAandBC       |
| 09H   | norAxorBC  | A?norBC:xnorBC | C?xorBA:norBA  | A?!B:xnorBC    | B?xorAC:norAC  | A?!C:xnorBC    | xnorABC         | A?nandBC:xnorBC |
| 0AH   | andCIA     | A?norBC:C      | andCnandBA     | A?!B:C         | C?!A:andBA     | xorCA          | xorCandBA       | A?nandBC:C      |
| 0BH   | C?!A:norBA | C?!A:!B        | C?nandBA:norBA | C?nandBA:!B    | B?xorAC:!A     | B?xorAC:nandAC | C?nandBA:xnorBA | nandBxnorAC     |
| 0CH   | andB!A     | A?norBC:B      | B?!A:andAC     | xorBA          | andBnandAC     | A?!C:B         | xorBandAC       | A?nandBC:B      |
| 0DH   | B?!A:norAC | B?!A:!C        | B?!A:xnorAC    | C?xorBA:nandBA | B?nandAC:norAC | B?nandAC:!C    | B?nandAC:xnorAC | nandCxnorBA     |
| 0EH   | norAnorBC  | xorAorBC       | B?!A:C         | A?!B:orBC      | C?!A:B         | A?!C:orBC      | B?nandAC:C      | A?nandBC:orBC   |
| 0FH   | !A         | nandAorBC      | C?nandBA:!A    | nandBA         | B?nandAC:!A    | nandCA         | nandAxnorBC     | nandABC         |

Table 5-2 shows the half of 256-entry map corresponding to column index values 8:15.

Table 5-2. Upper 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

| Imm   | [7:4]                |                        |                       |                      |                       |                      |                      |                   |
|-------|----------------------|------------------------|-----------------------|----------------------|-----------------------|----------------------|----------------------|-------------------|
| [3:0] | 08H                  | 09H                    | 0AH                   | 0BH                  | 0CH                   | 0DH                  | 0EH                  | 0FH               |
| 00H   | <i>andABC</i>        | <i>andAxnorBC</i>      | <i>andCA</i>          | <i>B?andAC:A</i>     | <i>andBA</i>          | <i>C?andBA:A</i>     | <i>andAorBC</i>      | <i>A</i>          |
| 01H   | <i>A?andBC:norBC</i> | <i>B?andAC:!C</i>      | <i>A?C:norBC</i>      | <i>C?A:!B</i>        | <i>A?B:norBC</i>      | <i>B?A:!C</i>        | <i>xnorAorBC</i>     | <i>orAnorBC</i>   |
| 02H   | <i>andCxnorBA</i>    | <i>B?andAC:xorAC</i>   | <i>B?andAC:C</i>      | <i>B?andAC:orAC</i>  | <i>C?xnorBA:andBA</i> | <i>B?A:xorAC</i>     | <i>B?A:C</i>         | <i>B?A:orAC</i>   |
| 03H   | <i>A?andBC:!B</i>    | <i>xnorBandAC</i>      | <i>A?C:!B</i>         | <i>nandBnandAC</i>   | <i>xnorBA</i>         | <i>B?A:nandAC</i>    | <i>A?orBC:!B</i>     | <i>orA!B</i>      |
| 04H   | <i>andBxnorAC</i>    | <i>C?andBA:xorBA</i>   | <i>B?xnorAC:andAC</i> | <i>B?xnorAC:A</i>    | <i>C?andBA:B</i>      | <i>C?andBA:orBA</i>  | <i>C?A:B</i>         | <i>C?A:orBA</i>   |
| 05H   | <i>A?andBC:!C</i>    | <i>xnorCandBA</i>      | <i>xnorCA</i>         | <i>C?A:nandBA</i>    | <i>A?B:!C</i>         | <i>nandCnandBA</i>   | <i>A?orBC:!C</i>     | <i>orA!C</i>      |
| 06H   | <i>A?andBC:xorBC</i> | <i>xorABC</i>          | <i>A?C:xorBC</i>      | <i>B?xnorAC:orAC</i> | <i>A?B:xorBC</i>      | <i>C?xnorBA:orBA</i> | <i>A?orBC:xorBC</i>  | <i>orAxorBC</i>   |
| 07H   | <i>xnorAandBC</i>    | <i>A?xnorBC:nandBC</i> | <i>A?C:nandBC</i>     | <i>nandBxorAC</i>    | <i>A?B:nandBC</i>     | <i>nandCxorBA</i>    | <i>A?orBCnandBC</i>  | <i>orAnandBC</i>  |
| 08H   | <i>andCB</i>         | <i>A?xnorBC:andBC</i>  | <i>andCorAB</i>       | <i>B?C:A</i>         | <i>andBorAC</i>       | <i>C?B:A</i>         | <i>majorABC</i>      | <i>orAandBC</i>   |
| 09H   | <i>B?C:norAC</i>     | <i>xnorCB</i>          | <i>xnorCorBA</i>      | <i>C?orBA:!B</i>     | <i>xnorBorAC</i>      | <i>B?orAC:!C</i>     | <i>A?orBC:xnorBC</i> | <i>orAxnorBC</i>  |
| 0AH   | <i>A?andBC:C</i>     | <i>A?xnorBC:C</i>      | <i>C</i>              | <i>B?C:orAC</i>      | <i>A?B:C</i>          | <i>B?orAC:xorAC</i>  | <i>orCandBA</i>      | <i>orCA</i>       |
| 0BH   | <i>B?C:!A</i>        | <i>B?C:nandAC</i>      | <i>orCnorBA</i>       | <i>orC!B</i>         | <i>B?orAC:!A</i>      | <i>B?orAC:nandAC</i> | <i>orCxnorBA</i>     | <i>nandBnorAC</i> |
| 0CH   | <i>A?andBC:B</i>     | <i>A?xnorBC:B</i>      | <i>A?C:B</i>          | <i>C?orBA:xorBA</i>  | <i>B</i>              | <i>C?B:orBA</i>      | <i>orBandAC</i>      | <i>orBA</i>       |
| 0DH   | <i>C?B!A</i>         | <i>C?B:nandBA</i>      | <i>C?orBA:!A</i>      | <i>C?orBA:nandBA</i> | <i>orBnorAC</i>       | <i>orB!C</i>         | <i>orBxnorAC</i>     | <i>nandCnorBA</i> |
| 0EH   | <i>A?andBC:orBC</i>  | <i>A?xnorBC:orBC</i>   | <i>A?C:orBC</i>       | <i>orCxorBA</i>      | <i>A?B:orBC</i>       | <i>orBxorAC</i>      | <i>orCB</i>          | <i>orABC</i>      |
| 0FH   | <i>nandAnandBC</i>   | <i>nandAxorBC</i>      | <i>orCIA</i>          | <i>orCnandBA</i>     | <i>orB!A</i>          | <i>orBnandAC</i>     | <i>nandAnorBC</i>    | <i>TRUE</i>       |

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

Result\_imm8 = Table\_Lookup\_Entry( 0F0H, 0CCH, 0AAH)

Table\_Lookup\_Entry is the Boolean expression defined in Table 5-1 and Table 5-2.

## 5.2 INSTRUCTIONS (V)

Chapter 5 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (V). See also: Chapter 3, “Instruction Set Reference, A-L,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A; Chapter 4, “Instruction Set Reference, M-U,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2B; and Chapter 6, “Instruction Set Reference, W-Z,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2D.

## VADDPH—Add Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.NP.MAP5.WO 58 /r<br>VADDPH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Add packed FP16 value from xmm3/m128/<br>m16bcst to xmm2, and store result in xmm1<br>subject to writemask k1. |
| EVEX.256.NP.MAP5.WO 58 /r<br>VADDPH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Add packed FP16 value from ymm3/m256/<br>m16bcst to ymm2, and store result in ymm1<br>subject to writemask k1. |
| EVEX.512.NP.MAP5.WO 58 /r<br>VADDPH zmm1{k1}{z}, zmm2, zmm3/<br>m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Add packed FP16 value from zmm3/m512/<br>m16bcst to zmm2, and store result in zmm1<br>subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|---------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction adds packed FP16 values from source operands and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

### Operation

#### VADDPH (EVEX Encoded Versions) When SRC2 Operand is a Register

VL = 128, 256 or 512

KL := VL/16

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

    SET\_RM(EVEX.RC)

ELSE

    SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

    IF k1[j] OR \*no writemask\*:

        DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]

    ELSEIF \*zeroing\*:

        DEST.fp16[j] := 0

    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

#### VADDPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

    IF k1[j] OR \*no writemask\*:

        IF EVEX.b = 1:

            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[0]

        ELSE:

            DEST.fp16[j] := SRC1.fp16[j] + SRC2.fp16[j]

    ELSE IF \*zeroing\*:

        DEST.fp16[j] := 0

    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VADDPH \_\_m128h \_\_mm\_add\_ph (\_\_m128h a, \_\_m128h b);  
 VADDPH \_\_m128h \_\_mm\_mask\_add\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
 VADDPH \_\_m128h \_\_mm\_maskz\_add\_ph (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
 VADDPH \_\_m256h \_\_mm256\_add\_ph (\_\_m256h a, \_\_m256h b);  
 VADDPH \_\_m256h \_\_mm256\_mask\_add\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a, \_\_m256h b);  
 VADDPH \_\_m256h \_\_mm256\_maskz\_add\_ph (\_\_mmask16 k, \_\_m256h a, \_\_m256h b);  
 VADDPH \_\_m512h \_\_mm512\_add\_ph (\_\_m512h a, \_\_m512h b);  
 VADDPH \_\_m512h \_\_mm512\_add\_ph (\_\_m512h a, \_\_m512h b);  
 VADDPH \_\_m512h \_\_mm512\_mask\_add\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, \_\_m512h b);  
 VADDPH \_\_m512h \_\_mm512\_maskz\_add\_ph (\_\_mmask32 k, \_\_m512h a, \_\_m512h b);  
 VADDPH \_\_m512h \_\_mm512\_add\_round\_ph (\_\_m512h a, \_\_m512h b, int rounding);  
 VADDPH \_\_m512h \_\_mm512\_mask\_add\_round\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, \_\_m512h b, int rounding);  
 VADDPH \_\_m512h \_\_mm512\_maskz\_add\_round\_ph (\_\_mmask32 k, \_\_m512h a, \_\_m512h b, int rounding);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

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



## VADDSH—Add Scalar FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F3.MAP5.W0 58 /r<br>VADDSH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Add the low FP16 value from xmm3/m16 to xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction adds the low FP16 value from the source operands and stores the FP16 result in the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

#### VADDSH (EVEX Encoded Versions)

IF EVEX.b = 1 and SRC2 is a register:

```
SET_RM(EVEX.RC)
```

ELSE

```
SET_RM(MXCSR.RC)
```

IF k1[0] OR \*no writemask\*:

```
DEST.fp16[0] := SRC1.fp16[0] + SRC2.fp16[0]
```

ELSEIF \*zeroing\*:

```
DEST.fp16[0] := 0
```

// else dest.fp16[0] remains unchanged

```
DEST[127:16] := SRC1[127:16]
```

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

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

```
VADDSH __m128h __mm_add_round_sh (__m128h a, __m128h b, int rounding);
```

```
VADDSH __m128h __mm_mask_add_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
```

```
VADDSH __m128h __mm_maskz_add_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
```

```
VADDSH __m128h __mm_add_sh (__m128h a, __m128h b);
```

```
VADDSH __m128h __mm_mask_add_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
```

```
VADDSH __m128h __mm_maskz_add_sh (__mmask8 k, __m128h a, __m128h b);
```

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

## VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

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

### Instruction Operand Encoding

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

### Description

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

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

**Operation****VALIGND (EVEX Encoded Versions)**

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

```

IF (SRC2 *is memory*) (AND EVEX.b = 1)
  THEN
    FOR j := 0 TO KL-1
      i := j * 32
      src[i+31:i] := SRC2[31:0]
    ENDFOR;
  ELSE src := SRC2
FI
; Concatenate sources
tmp[VL-1:0] := src[VL-1:0]
tmp[2VL-1:VL] := SRC1[VL-1:0]
; Shift right doubleword elements
IF VL = 128
  THEN SHIFT = imm8[1:0]
  ELSE
    IF VL = 256
      THEN SHIFT = imm8[2:0]
      ELSE SHIFT = imm8[3:0]
    FI
FI;
tmp[2VL-1:0] := tmp[2VL-1:0] >> (32*SHIFT)
; Apply writemask
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := tmp[i+31:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+31:i] := 0
        FI
      FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

```

**VALIGNQ (EVEX Encoded Versions)**

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

```

IF (SRC2 *is memory*) (AND EVEX.b = 1)
  THEN
    FOR j := 0 TO KL-1
      i := j * 64
      src[i+63:i] := SRC2[63:0]
    ENDFOR;
  ELSE src := SRC2
FI
; Concatenate sources
tmp[VL-1:0] := src[VL-1:0]
tmp[2VL-1:VL] := SRC1[VL-1:0]
; Shift right quadword elements

```

```

IF VL = 128
    THEN SHIFT = imm8[0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[1:0]
            ELSE SHIFT = imm8[2:0]
        FI
    FI;
tmp[2VL-1:0] := tmp[2VL-1:0] >> (64*SHIFT)
; Apply writemask
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] := tmp[i+63:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+63:i] := 0
            FI
        FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

```

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

```

VALIGND __m512i __mm512_alignr_epi32( __m512i a, __m512i b, int cnt);
VALIGND __m512i __mm512_mask_alignr_epi32( __m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i __mm512_maskz_alignr_epi32( __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i __mm256_mask_alignr_epi32( __m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i __mm256_maskz_alignr_epi32( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i __mm_mask_alignr_epi32( __m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i __mm_maskz_alignr_epi32( __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m512i __mm512_alignr_epi64( __m512i a, __m512i b, int cnt);
VALIGNQ __m512i __mm512_mask_alignr_epi64( __m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i __mm512_maskz_alignr_epi64( __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i __mm256_mask_alignr_epi64( __m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i __mm256_maskz_alignr_epi64( __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i __mm_mask_alignr_epi64( __m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i __mm_maskz_alignr_epi64( __mmask8 k, __m128i a, __m128i b, int cnt);

```

### Exceptions

See Table 2-50, "Type E4NF Class Exception Conditions."

## VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 65 /r<br>VBLENDMPD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | A          | V/V                          | AVX512VL<br>AVX512F      | Blend double precision vector xmm2 and double precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.        |
| EVEX.256.66.0F38.W1 65 /r<br>VBLENDMPD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | A          | V/V                          | AVX512VL<br>AVX512F      | Blend double precision vector ymm2 and double precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.        |
| EVEX.512.66.0F38.W1 65 /r<br>VBLENDMPD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst | A          | V/V                          | AVX512F                  | Blend double precision vector zmm2 and double precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.        |
| EVEX.128.66.0F38.W0 65 /r<br>VBLENDMPS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst | A          | V/V                          | AVX512VL<br>AVX512F      | Blend single precision vector xmm2 and single precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.        |
| EVEX.256.66.0F38.W0 65 /r<br>VBLENDMPS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst | A          | V/V                          | AVX512VL<br>AVX512F      | Blend single precision vector ymm2 and single precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.        |
| EVEX.512.66.0F38.W0 65 /r<br>VBLENDMPS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst | A          | V/V                          | AVX512F                  | Blend single precision vector zmm2 and single precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1. |

### Instruction Operand Encoding

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

### Description

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

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

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

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

### Operation

#### VBLENDMPD (EVEX Encoded Versions)

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

FOR j := 0 TO KL-1

  i := j \* 64

  IF k1[j] OR \*no controlmask\*

    THEN

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

        THEN

          DEST[i+63:i] := SRC2[63:0]

        ELSE

          DEST[i+63:i] := SRC2[i+63:i]

```

    FI;
ELSE
    IF *merging-masking*           ; merging-masking
        THEN DEST[i+63:i] := SRC1[i+63:i]
    ELSE                             ; zeroing-masking
        DEST[i+63:i] := 0
    FI;
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

**VBLENDMPS (EVEX Encoded Versions)**

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

FOR j := 0 TO KL-1

```

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

```

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

```

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

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions."

## VBROADCAST—Load with Broadcast Floating-Point Data

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| VEX.128.66.0F38.W0 18 /r<br>VBROADCASTSS xmm1, m32                     | A         | V/V                          | AVX                      | Broadcast single precision floating-point element in mem to four locations in xmm1.                              |
| VEX.256.66.0F38.W0 18 /r<br>VBROADCASTSS ymm1, m32                     | A         | V/V                          | AVX                      | Broadcast single precision floating-point element in mem to eight locations in ymm1.                             |
| VEX.256.66.0F38.W0 19 /r<br>VBROADCASTSD ymm1, m64                     | A         | V/V                          | AVX                      | Broadcast double precision floating-point element in mem to four locations in ymm1.                              |
| VEX.256.66.0F38.W0 1A /r<br>VBROADCASTF128 ymm1, m128                  | A         | V/V                          | AVX                      | Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.                               |
| VEX.128.66.0F38.W0 18/r<br>VBROADCASTSS xmm1, xmm2                     | A         | V/V                          | AVX2                     | Broadcast the low single precision floating-point element in the source operand to four locations in xmm1.       |
| VEX.256.66.0F38.W0 18 /r<br>VBROADCASTSS ymm1, xmm2                    | A         | V/V                          | AVX2                     | Broadcast low single precision floating-point element in the source operand to eight locations in ymm1.          |
| VEX.256.66.0F38.W0 19 /r<br>VBROADCASTSD ymm1, xmm2                    | A         | V/V                          | AVX2                     | Broadcast low double precision floating-point element in the source operand to four locations in ymm1.           |
| EVEX.256.66.0F38.W1 19 /r<br>VBROADCASTSD ymm1 {k1}{z},<br>xmm2/m64    | B         | V/V                          | AVX512VL<br>AVX512F      | Broadcast low double precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.  |
| EVEX.512.66.0F38.W1 19 /r<br>VBROADCASTSD zmm1 {k1}{z},<br>xmm2/m64    | B         | V/V                          | AVX512F                  | Broadcast low double precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 19 /r<br>VBROADCASTF32X2 ymm1 {k1}{z},<br>xmm2/m64 | C         | V/V                          | AVX512VL<br>AVX512DQ     | Broadcast two single precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.      |
| EVEX.512.66.0F38.W0 19 /r<br>VBROADCASTF32X2 zmm1 {k1}{z},<br>xmm2/m64 | C         | V/V                          | AVX512DQ                 | Broadcast two single precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.      |
| EVEX.128.66.0F38.W0 18 /r<br>VBROADCASTSS xmm1 {k1}{z},<br>xmm2/m32    | B         | V/V                          | AVX512VL<br>AVX512F      | Broadcast low single precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.   |
| EVEX.256.66.0F38.W0 18 /r<br>VBROADCASTSS ymm1 {k1}{z},<br>xmm2/m32    | B         | V/V                          | AVX512VL<br>AVX512F      | Broadcast low single precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.   |
| EVEX.512.66.0F38.W0 18 /r<br>VBROADCASTSS zmm1 {k1}{z},<br>xmm2/m32    | B         | V/V                          | AVX512F                  | Broadcast low single precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.   |
| EVEX.256.66.0F38.W0 1A /r<br>VBROADCASTF32X4 ymm1 {k1}{z},<br>m128     | D         | V/V                          | AVX512VL<br>AVX512F      | Broadcast 128 bits of 4 single precision floating-point data in mem to locations in ymm1 using writemask k1.     |
| EVEX.512.66.0F38.W0 1A /r<br>VBROADCASTF32X4 zmm1 {k1}{z},<br>m128     | D         | V/V                          | AVX512F                  | Broadcast 128 bits of 4 single precision floating-point data in mem to locations in zmm1 using writemask k1.     |
| EVEX.256.66.0F38.W1 1A /r<br>VBROADCASTF64X2 ymm1 {k1}{z},<br>m128     | C         | V/V                          | AVX512VL<br>AVX512DQ     | Broadcast 128 bits of 2 double precision floating-point data in mem to locations in ymm1 using writemask k1.     |

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.512.66.0F38.W1 1A /r<br>VBROADCASTF64X2 zmm1 {k1}{z},<br>m128 | C         | V/V                          | AVX512DQ                 | Broadcast 128 bits of 2 double precision floating-point data in mem to locations in zmm1 using writemask k1. |
| EVEX.512.66.0F38.W0 1B /r<br>VBROADCASTF32X8 zmm1 {k1}{z},<br>m256 | E         | V/V                          | AVX512DQ                 | Broadcast 256 bits of 8 single precision floating-point data in mem to locations in zmm1 using writemask k1. |
| EVEX.512.66.0F38.W1 1B /r<br>VBROADCASTF64X4 zmm1 {k1}{z},<br>m256 | D         | V/V                          | AVX512F                  | Broadcast 256 bits of 4 double precision floating-point data in mem to locations in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | N/A           | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| C     | Tuple2        | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| D     | Tuple4        | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| E     | Tuple8        | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

#### 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/VBROADCASTF32X8/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

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

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF32X8 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.



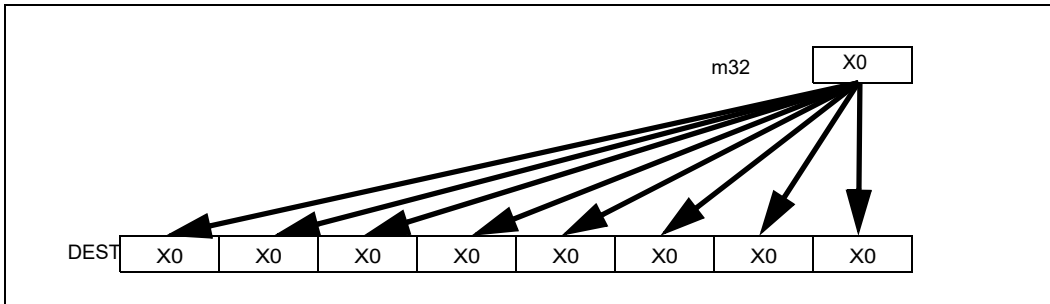


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

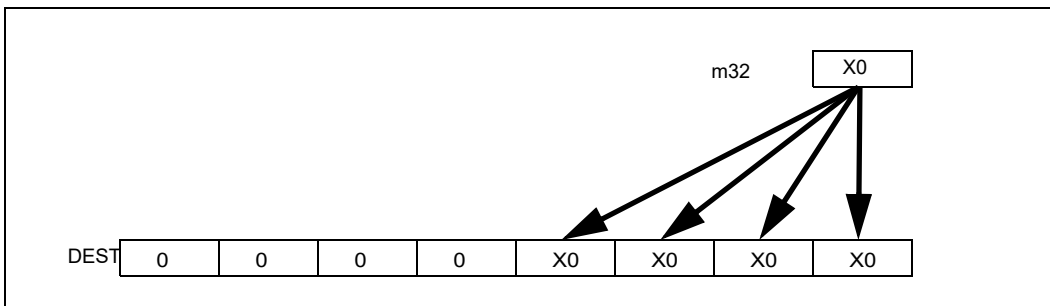


Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

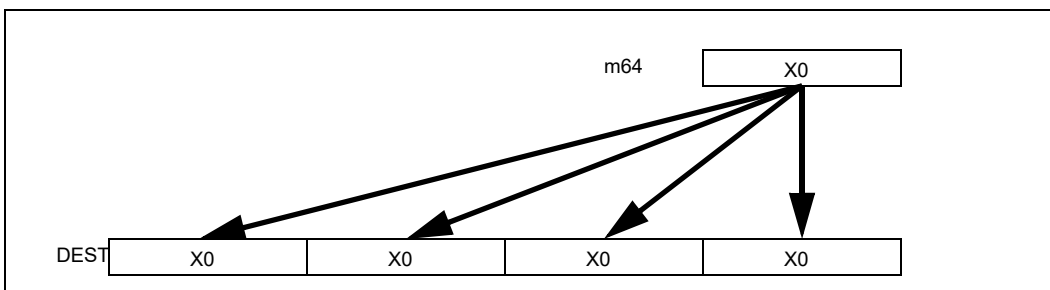


Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

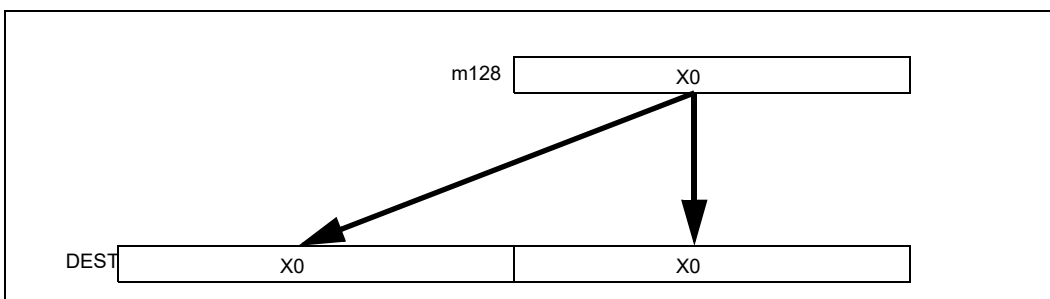


Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)

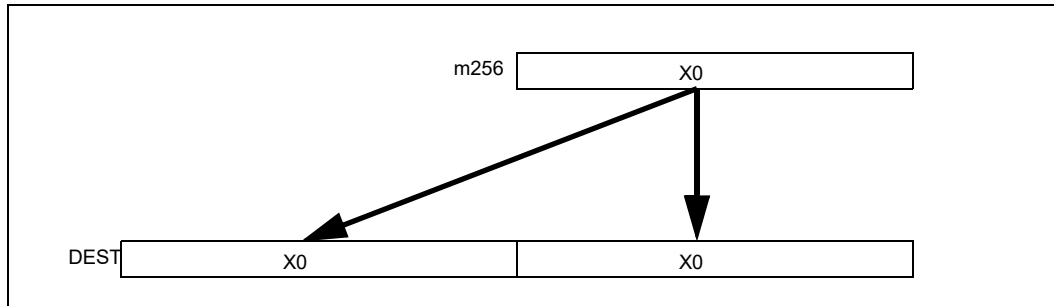


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

### Operation

#### VBROADCASTSS (128-bit Version VEX and Legacy)

```
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
```

#### VBROADCASTSS (VEX.256 Encoded Version)

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

#### VBROADCASTSS (EVEX Encoded Versions)

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

FOR j := 0 TO KL-1

  i := j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] := SRC[31:0]

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+31:i] := 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VBROADCASTSD (VEX.256 Encoded Version)**

```
temp := SRC[63:0]
DEST[63:0] := temp
DEST[127:64] := temp
DEST[191:128] := temp
DEST[255:192] := temp
DEST[MAXVL-1:256] := 0
```

**VBROADCASTSD (EVEX Encoded Versions)**

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

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

**VBROADCASTF32x2 (EVEX Encoded Versions)**

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

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

**VBROADCASTF128 (VEX.256 Encoded Version)**

```
temp := SRC[127:0]
DEST[127:0] := temp
DEST[255:128] := temp
DEST[MAXVL-1:256] := 0
```

**VBROADCASTF32X4 (EVEX Encoded Versions)**

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

FOR j := 0 TO KL-1

i := j \* 32

n := (j modulo 4) \* 32

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

THEN DEST[i+31:i] := SRC[n+31:n]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VBROADCASTF64X2 (EVEX Encoded Versions)**

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

FOR j := 0 TO KL-1

i := j \* 64

n := (j modulo 2) \* 64

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

THEN DEST[i+63:i] := SRC[n+63:n]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] = 0

FI

FI;

ENDFOR;

**VBROADCASTF32X8 (EVEX.U1.512 Encoded Version)**

FOR j := 0 TO 15

i := j \* 32

n := (j modulo 8) \* 32

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

THEN DEST[i+31:i] := SRC[n+31:n]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VBROADCASTF64X4 (EVEX.512 Encoded Version)**

```

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

```

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

```

VBROADCASTF32x2 __m512 __mm512_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m512 __mm512_mask_broadcast_f32x2( __m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 __mm512_maskz_broadcast_f32x2( __mmask16 k, __m128 a);
VBROADCASTF32x2 __m256 __mm256_broadcast_f32x2( __m128 a);
VBROADCASTF32x2 __m256 __mm256_mask_broadcast_f32x2( __m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m256 __mm256_maskz_broadcast_f32x2( __mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 __mm512_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m512 __mm512_mask_broadcast_f32x4( __m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 __mm512_maskz_broadcast_f32x4( __mmask16 k, __m128 a);
VBROADCASTF32x4 __m256 __mm256_broadcast_f32x4( __m128 a);
VBROADCASTF32x4 __m256 __mm256_mask_broadcast_f32x4( __m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x4 __m256 __mm256_maskz_broadcast_f32x4( __mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 __mm512_broadcast_f32x8( __m256 a);
VBROADCASTF32x8 __m512 __mm512_mask_broadcast_f32x8( __m512 s, __mmask16 k, __m256 a);
VBROADCASTF32x8 __m512 __mm512_maskz_broadcast_f32x8( __mmask16 k, __m256 a);
VBROADCASTF64x2 __m512d __mm512_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m512d __mm512_mask_broadcast_f64x2( __m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d __mm512_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d __mm256_broadcast_f64x2( __m128d a);
VBROADCASTF64x2 __m256d __mm256_mask_broadcast_f64x2( __m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d __mm256_maskz_broadcast_f64x2( __mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d __mm512_broadcast_f64x4( __m256d a);
VBROADCASTF64x4 __m512d __mm512_mask_broadcast_f64x4( __m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512d __mm512_maskz_broadcast_f64x4( __mmask8 k, __m256d a);
VBROADCASTSD __m512d __mm512_broadcastsd_pd( __m128d a);
VBROADCASTSD __m512d __mm512_mask_broadcastsd_pd( __m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d __mm512_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSD __m256d __mm256_broadcastsd_pd( __m128d a);
VBROADCASTSD __m256d __mm256_mask_broadcastsd_pd( __m256d s, __mmask8 k, __m128d a);
VBROADCASTSD __m256d __mm256_maskz_broadcastsd_pd( __mmask8 k, __m128d a);
VBROADCASTSD __m256d __mm256_broadcast_sd(double *a);
VBROADCASTSS __m512 __mm512_broadcastss_ps( __m128 a);
VBROADCASTSS __m512 __mm512_mask_broadcastss_ps( __m512 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512 __mm512_maskz_broadcastss_ps( __mmask16 k, __m128 a);
VBROADCASTSS __m256 __mm256_broadcastss_ps( __m128 a);
VBROADCASTSS __m256 __mm256_mask_broadcastss_ps( __m256 s, __mmask8 k, __m128 a);
VBROADCASTSS __m256 __mm256_maskz_broadcastss_ps( __mmask8 k, __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.                            |

## VCMPPH—Compare Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.NP.OF3A.W0 C2 /r /ib<br>VCMPPH k1{k2}, xmm2, xmm3/<br>m128/m16bcst, imm8       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Compare packed FP16 values in xmm3/m128/<br>m16bcst and xmm2 using bits 4:0 of imm8 as a<br>comparison predicate subject to writemask k2, and<br>store the result in mask register k1. |
| EVEX.256.NP.OF3A.W0 C2 /r /ib<br>VCMPPH k1{k2}, ymm2, ymm3/<br>m256/m16bcst, imm8       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Compare packed FP16 values in ymm3/m256/<br>m16bcst and ymm2 using bits 4:0 of imm8 as a<br>comparison predicate subject to writemask k2, and<br>store the result in mask register k1. |
| EVEX.512.NP.OF3A.W0 C2 /r /ib<br>VCMPPH k1{k2}, zmm2, zmm3/<br>m512/m16bcst {sae}, imm8 | A         | V/V                          | AVX512-FP16             | Compare packed FP16 values in zmm3/m512/<br>m16bcst and zmm2 using bits 4:0 of imm8 as a<br>comparison predicate subject to writemask k2, and<br>store the result in mask register k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|---------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | imm8 (r)  |

### Description

This instruction compares packed FP16 values from source operands and stores the result in the destination mask operand. The comparison predicate operand (immediate byte bits 4:0) specifies the type of comparison performed on each of the pairs of packed values. The destination elements are updated according to the writemask.

### Operation

CASE (imm8 & 0x1F) OF  
 0: CMP\_OPERATOR := EQ\_OQ;  
 1: CMP\_OPERATOR := LT\_OS;  
 2: CMP\_OPERATOR := LE\_OS;  
 3: CMP\_OPERATOR := UNORD\_Q;  
 4: CMP\_OPERATOR := NEQ\_UQ;  
 5: CMP\_OPERATOR := NLT\_US;  
 6: CMP\_OPERATOR := NLE\_US;  
 7: CMP\_OPERATOR := ORD\_Q;  
 8: CMP\_OPERATOR := EQ\_UQ;  
 9: CMP\_OPERATOR := NGE\_US;  
 10: CMP\_OPERATOR := NGT\_US;  
 11: CMP\_OPERATOR := FALSE\_OQ;  
 12: CMP\_OPERATOR := NEQ\_OQ;  
 13: CMP\_OPERATOR := GE\_OS;  
 14: CMP\_OPERATOR := GT\_OS;  
 15: CMP\_OPERATOR := TRUE\_UQ;  
 16: CMP\_OPERATOR := EQ\_OS;  
 17: CMP\_OPERATOR := LT\_OQ;  
 18: CMP\_OPERATOR := LE\_OQ;  
 19: CMP\_OPERATOR := UNORD\_S;  
 20: CMP\_OPERATOR := NEQ\_US;  
 21: CMP\_OPERATOR := NLT\_UQ;  
 22: CMP\_OPERATOR := NLE\_UQ;  
 23: CMP\_OPERATOR := ORD\_S;

```

24: CMP_OPERATOR := EQ_US;
25: CMP_OPERATOR := NGE_UQ;
26: CMP_OPERATOR := NGT_UQ;
27: CMP_OPERATOR := FALSE_OS;
28: CMP_OPERATOR := NEQ_OS;
29: CMP_OPERATOR := GE_OQ;
30: CMP_OPERATOR := GT_OQ;
31: CMP_OPERATOR := TRUE_US;
ESAC

```

**VCMPPH (EVEX Encoded Versions)**

VL = 128, 256 or 512

KL := VL/16

```

FOR j := 0 TO KL-1:
  IF k2[j] OR *no writemask*:
    IF EVEX.b = 1:
      tsrc2 := SRC2.fp16[0]
    ELSE:
      tsrc2 := SRC2.fp16[j]
    DEST.bit[j] := SRC1.fp16[j] CMP_OPERATOR tsrc2
  ELSE
    DEST.bit[j] := 0

```

DEST[MAXKL-1:KL] := 0

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

```

VCMPPH __mmask8 _mm_cmp_ph_mask (__m128h a, __m128h b, const int imm8);
VCMPPH __mmask8 _mm_mask_cmp_ph_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8);
VCMPPH __mmask16 _mm256_cmp_ph_mask (__m256h a, __m256h b, const int imm8);
VCMPPH __mmask16 _mm256_mask_cmp_ph_mask (__mmask16 k1, __m256h a, __m256h b, const int imm8);
VCMPPH __mmask32 _mm512_cmp_ph_mask (__m512h a, __m512h b, const int imm8);
VCMPPH __mmask32 _mm512_mask_cmp_ph_mask (__mmask32 k1, __m512h a, __m512h b, const int imm8);
VCMPPH __mmask32 _mm512_cmp_round_ph_mask (__m512h a, __m512h b, const int imm8, const int sae);
VCMPPH __mmask32 _mm512_mask_cmp_round_ph_mask (__mmask32 k1, __m512h a, __m512h b, const int imm8, const int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

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



## VCMPSH—Compare Scalar FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|--|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.F3.0F3A.W0 C2 /r /ib<br>VCMPSH k1{k2}, xmm2, xmm3/m16<br>{sae}, imm8 | A         | V/V                          | AVX512-FP16           | Compare low FP16 values in xmm3/m16 and xmm2 using bits 4:0 of imm8 as a comparison predicate subject to writemask k2, and store the result in mask register k1. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | imm8 (r)  |

### Description

This instruction compares the FP16 values from the lowest element of the source operands and stores the result in the destination mask operand. The comparison predicate operand (immediate byte bits 4:0) specifies the type of comparison performed on the pair of packed FP16 values. The low destination bit is updated according to the writemask. Bits MAXKL-1:1 of the destination operand are zeroed.

### Operation

CASE (imm8 & 0x1F) OF  
 0: CMP\_OPERATOR := EQ\_OQ;  
 1: CMP\_OPERATOR := LT\_OS;  
 2: CMP\_OPERATOR := LE\_OS;  
 3: CMP\_OPERATOR := UNORD\_Q;  
 4: CMP\_OPERATOR := NEQ\_UQ;  
 5: CMP\_OPERATOR := NLT\_US;  
 6: CMP\_OPERATOR := NLE\_US;  
 7: CMP\_OPERATOR := ORD\_Q;  
 8: CMP\_OPERATOR := EQ\_UQ;  
 9: CMP\_OPERATOR := NGE\_US;  
 10: CMP\_OPERATOR := NGT\_US;  
 11: CMP\_OPERATOR := FALSE\_OQ;  
 12: CMP\_OPERATOR := NEQ\_OQ;  
 13: CMP\_OPERATOR := GE\_OS;  
 14: CMP\_OPERATOR := GT\_OS;  
 15: CMP\_OPERATOR := TRUE\_UQ;  
 16: CMP\_OPERATOR := EQ\_OS;  
 17: CMP\_OPERATOR := LT\_OQ;  
 18: CMP\_OPERATOR := LE\_OQ;  
 19: CMP\_OPERATOR := UNORD\_S;  
 20: CMP\_OPERATOR := NEQ\_US;  
 21: CMP\_OPERATOR := NLT\_UQ;  
 22: CMP\_OPERATOR := NLE\_UQ;  
 23: CMP\_OPERATOR := ORD\_S;  
 24: CMP\_OPERATOR := EQ\_US;  
 25: CMP\_OPERATOR := NGE\_UQ;  
 26: CMP\_OPERATOR := NGT\_UQ;  
 27: CMP\_OPERATOR := FALSE\_OS;  
 28: CMP\_OPERATOR := NEQ\_OS;  
 29: CMP\_OPERATOR := GE\_OQ;  
 30: CMP\_OPERATOR := GT\_OQ;

```
31: CMP_OPERATOR := TRUE_US;
ESAC
```

### VCMPSH (EVEX Encoded Versions)

```
IF k2[0] OR *no writemask*:
    DEST.bit[0] := SRC1.fp16[0] CMP_OPERATOR SRC2.fp16[0]
ELSE
    DEST.bit[0] := 0

DEST[MAXKL-1:1] := 0
```

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

```
VCMPSH __mmask8 _mm_cmp_round_sh_mask (__m128h a, __m128h b, const int imm8, const int sae);
VCMPSH __mmask8 _mm_mask_cmp_round_sh_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8, const int sae);
VCMPSH __mmask8 _mm_cmp_sh_mask (__m128h a, __m128h b, const int imm8);
VCMPSH __mmask8 _mm_mask_cmp_sh_mask (__mmask8 k1, __m128h a, __m128h b, const int imm8);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal.

### Other Exceptions

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

## VCOMISH—Compare Scalar Ordered FP16 Values and Set EFLAGS

| Opcode/<br>Instruction                                     | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.NP.MAP5.W0 2F /r<br>VCOMISH xmm1, xmm2/m16 {sae} | A         | V/V                          | AVX512-FP16           | Compare low FP16 values in xmm1 and xmm2/<br>m16, and set the EFLAGS flags accordingly. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------|---------------|---------------|-----------|-----------|
| A     | Scalar | ModRM:reg (r) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction compares the FP16 values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

The VCOMISH instruction differs from the VUCOMISH instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The VUCOMISH instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

### Operation

#### VCOMISH SRC1, SRC2

RESULT := OrderedCompare(SRC1.fp16[0],SRC2.fp16[0])

IF RESULT is UNORDERED:

ZF, PF, CF := 1, 1, 1

ELSE IF RESULT is GREATER\_THAN:

ZF, PF, CF := 0, 0, 0

ELSE IF RESULT is LESS\_THAN:

ZF, PF, CF := 0, 0, 1

ELSE: // RESULT is EQUALS

ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0

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

VCOMISH int \_\_mm\_comi\_round\_sh (\_\_m128h a, \_\_m128h b, const int imm8, const int sae);

VCOMISH int \_\_mm\_comi\_sh (\_\_m128h a, \_\_m128h b, const int imm8);

VCOMISH int \_\_mm\_comieq\_sh (\_\_m128h a, \_\_m128h b);

VCOMISH int \_\_mm\_comige\_sh (\_\_m128h a, \_\_m128h b);

VCOMISH int \_\_mm\_comigt\_sh (\_\_m128h a, \_\_m128h b);

VCOMISH int \_\_mm\_comile\_sh (\_\_m128h a, \_\_m128h b);

VCOMISH int \_\_mm\_comilt\_sh (\_\_m128h a, \_\_m128h b);

VCOMISH int \_\_mm\_comineq\_sh (\_\_m128h a, \_\_m128h b);

### SIMD Floating-Point Exceptions

Invalid, Denormal.

**Other Exceptions**

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

## VCOMPRESSPD—Store Sparse Packed Double Precision Floating-Point Values Into Dense Memory

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W1 8A /r<br>VCOMPRESSPD xmm1/m128 {k1}{z},<br>xmm2 | A          | V/V                          | AVX512VL<br>AVX512F      | Compress packed double precision floating-point values from xmm2 to xmm1/m128 using writemask k1.    |
| EVEX.256.66.0F38.W1 8A /r<br>VCOMPRESSPD ymm1/m256 {k1}{z},<br>ymm2 | A          | V/V                          | AVX512VL<br>AVX512F      | Compress packed double precision floating-point values from ymm2 to ymm1/m256 using writemask k1.    |
| EVEX.512.66.0F38.W1 8A /r<br>VCOMPRESSPD zmm1/m512 {k1}{z},<br>zmm2 | A          | V/V                          | AVX512F                  | Compress packed double precision floating-point values from zmm2 using control mask k1 to zmm1/m512. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

### 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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 8A /r<br>VCOMPRESSPS xmm1/m128 {k1}{z},<br>xmm2 | A          | V/V                          | AVX512VL<br>AVX512F      | Compress packed single precision floating-point values from xmm2 to xmm1/m128 using writemask k1.    |
| EVEX.256.66.0F38.W0 8A /r<br>VCOMPRESSPS ymm1/m256 {k1}{z},<br>ymm2 | A          | V/V                          | AVX512VL<br>AVX512F      | Compress packed single precision floating-point values from ymm2 to ymm1/m256 using writemask k1.    |
| EVEX.512.66.0F38.W0 8A /r<br>VCOMPRESSPS zmm1/m512 {k1}{z},<br>zmm2 | A          | V/V                          | AVX512F                  | Compress packed single precision floating-point values from zmm2 using control mask k1 to zmm1/m512. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

### Description

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

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

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

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

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

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

### Operation

#### VCOMPRESSPS (EVEX Encoded Versions) Store Form

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

SIZE := 32

k := 0

FOR j := 0 TO KL-1

  i := j \* 32

  IF k1[j] OR \*no writemask\*

  THEN

    DEST[k+SIZE-1:k] := SRC[i+31:i]

    k := k + SIZE

  FI;

ENDFOR;

**VCOMPRESSPS (EVEX Encoded Versions) Reg-Reg Form**

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

SIZE := 32

k := 0

FOR j := 0 TO KL-1

i := j \* 32

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

THEN

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

k := k + SIZE

FI;

ENDFOR

IF \*merging-masking\*

THEN \*DEST[VL-1:k] remains unchanged\*

ELSE DEST[VL-1:k] := 0

FI

DEST[MAXVL-1:VL] := 0

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

VCOMPRESSPS \_\_m512 \_\_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.



## VCVTDQ2PH—Convert Packed Signed Doubleword Integers to Packed FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.MAP5.WO 5B /r<br>VCVTDQ2PH xmm1{k1}{z}, xmm2/<br>m128/m32bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed FP16 values, and store the result in xmm1 subject to writemask k1.       |
| EVEX.256.NP.MAP5.WO 5B /r<br>VCVTDQ2PH xmm1{k1}{z}, ymm2/<br>m256/m32bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed FP16 values, and store the result in xmm1 subject to writemask k1.     |
| EVEX.512.NP.MAP5.WO 5B /r<br>VCVTDQ2PH ymm1{k1}{z}, zmm2/<br>m512/m32bcst {er} | A         | V/V                          | AVX512-FP16             | Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed FP16 values, and store the result in ymm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts four, eight, or sixteen packed signed doubleword integers in the source operand to four, eight, or sixteen packed FP16 values in the destination operand.

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

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

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

### Operation

**VCVTDQ2PH DEST, SRC**

VL = 128, 256 or 512

KL := VL / 32

IF \*SRC is a register\* and (VL = 512) AND (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.dword[0]

ELSE

tsrc := SRC.dword[j]

DEST.fp16[j] := Convert\_integer32\_to\_fp16(tsrc)

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

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

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

```
VCVTDQ2PH __m256h __mm512_cvt_roundepi32_ph (__m512i a, int rounding);
VCVTDQ2PH __m256h __mm512_mask_cvt_roundepi32_ph (__m256h src, __mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m256h __mm512_maskz_cvt_roundepi32_ph (__mmask16 k, __m512i a, int rounding);
VCVTDQ2PH __m128h __mm_cvtepi32_ph (__m128i a);
VCVTDQ2PH __m128h __mm_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m128i a);
VCVTDQ2PH __m128h __mm_maskz_cvtepi32_ph (__mmask8 k, __m128i a);
VCVTDQ2PH __m128h __mm256_cvtepi32_ph (__m256i a);
VCVTDQ2PH __m128h __mm256_mask_cvtepi32_ph (__m128h src, __mmask8 k, __m256i a);
VCVTDQ2PH __m128h __mm256_maskz_cvtepi32_ph (__mmask8 k, __m256i a);
VCVTDQ2PH __m256h __mm512_cvtepi32_ph (__m512i a);
VCVTDQ2PH __m256h __mm512_mask_cvtepi32_ph (__m256h src, __mmask16 k, __m512i a);
VCVTDQ2PH __m256h __mm512_maskz_cvtepi32_ph (__mmask16 k, __m512i a);
```

### SIMD Floating-Point Exceptions

Overflow, Precision.

### Other Exceptions

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

## VCVTNE2PS2BF16—Convert Two Packed Single Data to One Packed BF16 Data

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.F2.0F38.W0 72 /r<br>VCVTNE2PS2BF16 xmm1{k1}{z},<br>xmm2, xmm3/m128/m32bcst | A         | V/V                          | AVX512VL<br>AVX512_BF16 | Convert packed single data from xmm2 and<br>xmm3/m128/m32bcst to packed BF16 data in<br>xmm1 with writemask k1. |
| EVEX.256.F2.0F38.W0 72 /r<br>VCVTNE2PS2BF16 ymm1{k1}{z},<br>ymm2, ymm3/m256/m32bcst | A         | V/V                          | AVX512VL<br>AVX512_BF16 | Convert packed single data from ymm2 and<br>ymm3/m256/m32bcst to packed BF16 data in<br>ymm1 with writemask k1. |
| EVEX.512.F2.0F38.W0 72 /r<br>VCVTNE2PS2BF16 zmm1{k1}{z},<br>zmm2, zmm3/m512/m32bcst | A         | V/V                          | AVX512F<br>AVX512_BF16  | Convert packed single data from zmm2 and<br>zmm3/m512/m32bcst to packed BF16 data in<br>zmm1 with writemask k1. |

### Instruction Operand Encoding

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

### 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 (__m128, __m128);`  
`VCVTNE2PS2BF16 __m128bh __mm_mask_cvtne2ps_pbh (__m128bh, __mmask8, __m128, __m128);`  
`VCVTNE2PS2BF16 __m128bh __mm_maskz_cvtne2ps_pbh (__mmask8, __m128, __m128);`  
`VCVTNE2PS2BF16 __m256bh __mm256_cvtne2ps_pbh (__m256, __m256);`  
`VCVTNE2PS2BF16 __m256bh __mm256_mask_cvtne2ps_pbh (__m256bh, __mmask16, __m256, __m256);`  
`VCVTNE2PS2BF16 __m256bh __mm256_maskz_cvtne2ps_pbh (__mmask16, __m256, __m256);`  
`VCVTNE2PS2BF16 __m512bh __mm512_cvtne2ps_pbh (__m512, __m512);`  
`VCVTNE2PS2BF16 __m512bh __mm512_mask_cvtne2ps_pbh (__m512bh, __mmask32, __m512, __m512);`  
`VCVTNE2PS2BF16 __m512bh __mm512_maskz_cvtne2ps_pbh (__mmask32, __m512, __m512);`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Table 2-50, “Type E4NF Class Exception Conditions.”

## VCVTNEPS2BF16—Convert Packed Single Data to Packed BF16 Data

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|-----------|------------------------------|-------------------------|--|
| EVEX.128.F3.0F38.W0 72 /r<br>VCVTNEPS2BF16 xmm1{k1}{z},<br>xmm2/m128/m32bcst | A         | V/V                          | AVX512VL<br>AVX512_BF16 | Convert packed single data from xmm2/m128 to packed BF16 data in xmm1 with writemask k1. |
| EVEX.256.F3.0F38.W0 72 /r<br>VCVTNEPS2BF16 xmm1{k1}{z},<br>ymm2/m256/m32bcst | A         | V/V                          | AVX512VL<br>AVX512_BF16 | Convert packed single data from ymm2/m256 to packed BF16 data in xmm1 with writemask k1. |
| EVEX.512.F3.0F38.W0 72 /r<br>VCVTNEPS2BF16 ymm1{k1}{z},<br>zmm2/m512/m32bcst | A         | V/V                          | AVX512F<br>AVX512_BF16  | Convert packed single data from zmm2/m512 to packed BF16 data in ymm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

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

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

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

### Operation

Define `convert_fp32_to_bfloat16(x)`:

IF x is zero or denormal:

`dest[15] := x[31] // sign preserving zero (denormal go to zero)`

`dest[14:0] := 0`

ELSE IF x is infinity:

`dest[15:0] := x[31:16]`

ELSE IF x is NAN:

`dest[15:0] := x[31:16] // truncate and set MSB of the mantissa to force QNAN`

`dest[6] := 1`

ELSE // normal number

`LSB := x[16]`

`rounding_bias := 0x00007FFF + LSB`

`temp[31:0] := x[31:0] + rounding_bias // integer add`

`dest[15:0] := temp[31:16]`

RETURN dest

**VCVTNEPS2BF16 dest, src**

VL = (128, 256, 512)

KL = VL/16

origdest := dest

FOR i := 0 to KL/2-1:

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

IF src is memory and evex.b == 1:

t := src.fp32[0]

ELSE:

t := src.fp32[ i ]

dest.word[i] := convert\_fp32\_to\_bfloat16(t)

ELSE IF \*zeroing\*:

dest.word[ i ] := 0

ELSE: // Merge masking, dest element unchanged

dest.word[ i ] := origdest.word[ i ]

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

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

VCVTNEPS2BF16 \_\_m128bh \_\_mm\_cvtneps\_pbh (\_\_m128);

VCVTNEPS2BF16 \_\_m128bh \_\_mm\_mask\_cvtneps\_pbh (\_\_m128bh, \_\_mmask8, \_\_m128);

VCVTNEPS2BF16 \_\_m128bh \_\_mm\_maskz\_cvtneps\_pbh (\_\_mmask8, \_\_m128);

VCVTNEPS2BF16 \_\_m128bh \_\_mm256\_cvtneps\_pbh (\_\_m256);

VCVTNEPS2BF16 \_\_m128bh \_\_mm256\_mask\_cvtneps\_pbh (\_\_m128bh, \_\_mmask8, \_\_m256);

VCVTNEPS2BF16 \_\_m128bh \_\_mm256\_maskz\_cvtneps\_pbh (\_\_mmask8, \_\_m256);

VCVTNEPS2BF16 \_\_m256bh \_\_mm512\_cvtneps\_pbh (\_\_m512);

VCVTNEPS2BF16 \_\_m256bh \_\_mm512\_mask\_cvtneps\_pbh (\_\_m256bh, \_\_mmask16, \_\_m512);

VCVTNEPS2BF16 \_\_m256bh \_\_mm512\_maskz\_cvtneps\_pbh (\_\_mmask16, \_\_m512);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions."

## VCVTPD2PH—Convert Packed Double Precision FP Values to Packed FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP5.W1 5A /r<br>VCVTPD2PH xmm1{k1}{z}, xmm2/<br>m128/m64bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two packed FP16 values, and store the result in xmm1 subject to writemask k1.     |
| EVEX.256.66.MAP5.W1 5A /r<br>VCVTPD2PH xmm1{k1}{z}, ymm2/<br>m256/m64bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four packed FP16 values, and store the result in xmm1 subject to writemask k1.   |
| EVEX.512.66.MAP5.W1 5A /r<br>VCVTPD2PH xmm1{k1}{z}, zmm2/<br>m512/m64bcst {er} | A         | V/V                          | AVX512-FP16             | Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight packed FP16 values, and store the result in ymm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

#### Description

This instruction converts two, four, or eight packed double precision floating-point values in the source operand (second operand) to two, four, or eight packed FP16 values 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.

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 broadcasts from a 64-bit memory location. The destination operand is a XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:128/64/32) of the corresponding destination are zeroed.

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

This instruction uses MXCSR.DAZ for handling FP64 inputs. FP16 outputs can be normal or denormal, and are not conditionally flushed to zero.

**Operation****VCVTPD2PH DEST, SRC**

VL = 128, 256 or 512

KL := VL / 64

IF \*SRC is a register\* and (VL = 512) AND (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.double[0]

ELSE

tsrc := SRC.double[j]

DEST.fp16[j] := Convert\_fp64\_to\_fp16(tsrc)

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0

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

VCVTPD2PH \_\_m128h \_\_mm512\_cvt\_roundpd\_ph (\_\_m512d a, int rounding);

VCVTPD2PH \_\_m128h \_\_mm512\_mask\_cvt\_roundpd\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m512d a, int rounding);

VCVTPD2PH \_\_m128h \_\_mm512\_maskz\_cvt\_roundpd\_ph (\_\_mmask8 k, \_\_m512d a, int rounding);

VCVTPD2PH \_\_m128h \_\_mm\_cvtspd\_ph (\_\_m128d a);

VCVTPD2PH \_\_m128h \_\_mm\_mask\_cvtspd\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128d a);

VCVTPD2PH \_\_m128h \_\_mm\_maskz\_cvtspd\_ph (\_\_mmask8 k, \_\_m128d a);

VCVTPD2PH \_\_m128h \_\_mm256\_cvtspd\_ph (\_\_m256d a);

VCVTPD2PH \_\_m128h \_\_mm256\_mask\_cvtspd\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m256d a);

VCVTPD2PH \_\_m128h \_\_mm256\_maskz\_cvtspd\_ph (\_\_mmask8 k, \_\_m256d a);

VCVTPD2PH \_\_m128h \_\_mm512\_cvtspd\_ph (\_\_m512d a);

VCVTPD2PH \_\_m128h \_\_mm512\_mask\_cvtspd\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m512d a);

VCVTPD2PH \_\_m128h \_\_mm512\_maskz\_cvtspd\_ph (\_\_mmask8 k, \_\_m512d a);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."



## VCVTPD2QQ—Convert Packed Double Precision Floating-Point Values to Packed Quadword Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F.W1 7B /r<br>VCVTPD2QQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed double precision floating-point values from xmm2/m128/m64bcst to two packed quadword integers in xmm1 with writemask k1.     |
| EVEX.256.66.0F.W1 7B /r<br>VCVTPD2QQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst     | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 with writemask k1.   |
| EVEX.512.66.0F.W1 7B /r<br>VCVTPD2QQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | A         | V/V                          | AVX512DQ                 | Convert eight packed double precision floating-point values from zmm2/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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, int r);
VCVTPD2QQ __m256i __mm256_mask_cvtpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i __mm256_maskz_cvtpd_epi64( __mmask8 k, __m256d a);
VCVTPD2QQ __m128i __mm_mask_cvtpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2QQ __m128i __mm_maskz_cvtpd_epi64( __mmask8 k, __m128d a);
VCVTPD2QQ __m256i __mm256_cvtpd_epi64( __m256d src)
VCVTPD2QQ __m128i __mm_cvtpd_epi64( __m128d src)

```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

EVEX-encoded instructions, see 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

| Opcode<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.OF.W1 79 /r<br>VCVTPD2UDQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | A         | V/V                          | AVX512VL<br>AVX512F      | Convert two packed double precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 subject to writemask k1.     |
| EVEX.256.OF.W1 79 /r<br>VCVTPD2UDQ xmm1 {k1}{z},<br>ymm2/m256/m64bcst     | A         | V/V                          | AVX512VL<br>AVX512F      | Convert four packed double precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 subject to writemask k1.   |
| EVEX.512.OF.W1 79 /r<br>VCVTPD2UDQ ymm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | A         | V/V                          | AVX512F                  | Convert eight packed double precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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, int r);
VCVTPD2UDQ __m128i __mm256_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i __mm256_maskz_cvtpd_epu32( __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i __mm_mask_cvtpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i __mm_maskz_cvtpd_epu32( __mmask8 k, __m128d a);

```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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

Additionally:

#UD If EVEX.vvvv != 1111B.

## VCVTPD2UQQ—Convert Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers

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

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

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

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

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

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

### Operation

#### VCVTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Register

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 64

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

THEN DEST[i+63:i] :=

Convert\_Double\_Precision\_Floating\_Point\_To\_UQuadInteger(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

```

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

```

### VCVTPD2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

```

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

```

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

```

VCVTPD2UQQ __m512i __mm512_cvtupd_epu64( __m512d a);
VCVTPD2UQQ __m512i __mm512_mask_cvtupd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i __mm512_maskz_cvtupd_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_mask_cvtupd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i __mm256_maskz_cvtupd_epu64( __mmask8 k, __m256d a);
VCVTPD2UQQ __m128i __mm_mask_cvtupd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i __mm_maskz_cvtupd_epu64( __mmask8 k, __m128d a);
VCVTPD2UQQ __m256i __mm256_cvtupd_epu64( __m256d src)
VCVTPD2UQQ __m128i __mm_cvtupd_epu64( __m128d src)

```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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

Additionally:

#UD                      If EVEX.vvvv != 1111B.

## VCVTPH2DQ—Convert Packed FP16 Values to Signed Doubleword Integers

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|-----------|------------------------------|-------------------------|--|
| EVEX.128.66.MAP5.W0 5B /r<br>VCVTPH2DQ xmm1{k1}{z}, xmm2/<br>m64/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed FP16 values in xmm2/m64/<br>m16bcst to four signed doubleword integers, and<br>store the result in xmm1 subject to writemask<br>k1.        |
| EVEX.256.66.MAP5.W0 5B /r<br>VCVTPH2DQ ymm1{k1}{z}, xmm2/<br>m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight signed doubleword<br>integers, and store the result in ymm1 subject to<br>writemask k1.     |
| EVEX.512.66.MAP5.W0 5B /r<br>VCVTPH2DQ zmm1{k1}{z}, ymm2/<br>m256/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Convert sixteen packed FP16 values in ymm2/<br>m256/m16bcst to sixteen signed doubleword<br>integers, and store the result in zmm1 subject to<br>writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Half  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to signed doubleword integers in 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 is returned.

The destination elements are updated according to the writemask.

### Operation

#### VCVTPH2DQ DEST, SRC

VL = 128, 256 or 512

KL := VL / 32

IF \*SRC is a register\* and (VL = 512) and (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.fp16[0]

ELSE

tsrc := SRC.fp16[j]

DEST.dword[j] := Convert\_fp16\_to\_integer32(tsrc)

ELSE IF \*zeroing\*:

DEST.dword[j] := 0

// else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTPH2DQ \_\_m512i \_\_mm512\_cvt\_roundph\_epi32 (\_\_m256h a, int rounding);  
 VCVTPH2DQ \_\_m512i \_\_mm512\_mask\_cvt\_roundph\_epi32 (\_\_m512i src, \_\_mmask16 k, \_\_m256h a, int rounding);  
 VCVTPH2DQ \_\_m512i \_\_mm512\_maskz\_cvt\_roundph\_epi32 (\_\_mmask16 k, \_\_m256h a, int rounding);  
 VCVTPH2DQ \_\_m128i \_\_mm\_cvtph\_epi32 (\_\_m128h a);  
 VCVTPH2DQ \_\_m128i \_\_mm\_mask\_cvtph\_epi32 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2DQ \_\_m128i \_\_mm\_maskz\_cvtph\_epi32 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2DQ \_\_m256i \_\_mm256\_cvtph\_epi32 (\_\_m128h a);  
 VCVTPH2DQ \_\_m256i \_\_mm256\_mask\_cvtph\_epi32 (\_\_m256i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2DQ \_\_m256i \_\_mm256\_maskz\_cvtph\_epi32 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2DQ \_\_m512i \_\_mm512\_cvtph\_epi32 (\_\_m256h a);  
 VCVTPH2DQ \_\_m512i \_\_mm512\_mask\_cvtph\_epi32 (\_\_m512i src, \_\_mmask16 k, \_\_m256h a);  
 VCVTPH2DQ \_\_m512i \_\_mm512\_maskz\_cvtph\_epi32 (\_\_mmask16 k, \_\_m256h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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



## VCVTPH2PD—Convert Packed FP16 Values to FP64 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.NP.MAP5.W0 5A /r<br>VCVTPH2PD xmm1{k1}{z}, xmm2/<br>m32/m16bcst        | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert packed FP16 values in xmm2/m32/<br>m16bcst to FP64 values, and store result in<br>xmm1 subject to writemask k1.  |
| EVEX.256.NP.MAP5.W0 5A /r<br>VCVTPH2PD ymm1{k1}{z}, xmm2/<br>m64/m16bcst        | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert packed FP16 values in xmm2/m64/<br>m16bcst to FP64 values, and store result in<br>ymm1 subject to writemask k1.  |
| EVEX.512.NP.MAP5.W0 5A /r<br>VCVTPH2PD zmm1{k1}{z}, xmm2/<br>m128/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Convert packed FP16 values in xmm2/m128/<br>m16bcst to FP64 values, and store result in<br>zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------|---------------|---------------|-----------|-----------|
| A     | Quarter | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values to FP64 values in the destination register. The destination elements are updated according to the writemask.

This instruction handles both normal and denormal FP16 inputs.

### Operation

**VCVTPH2PD DEST, SRC**

VL = 128, 256, or 512

KL := VL/64

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.fp16[0]

ELSE

tsrc := SRC.fp16[j]

DEST.fp64[j] := Convert\_fp16\_to\_fp64(tsrc)

ELSE IF \*zeroing\*:

DEST.fp64[j] := 0

// else dest.fp64[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTPH2PD \_\_m512d \_\_mm512\_cvt\_roundph\_pd (\_\_m128h a, int sae);  
 VCVTPH2PD \_\_m512d \_\_mm512\_mask\_cvt\_roundph\_pd (\_\_m512d src, \_\_mmask8 k, \_\_m128h a, int sae);  
 VCVTPH2PD \_\_m512d \_\_mm512\_maskz\_cvt\_roundph\_pd (\_\_mmask8 k, \_\_m128h a, int sae);  
 VCVTPH2PD \_\_m128d \_\_mm\_cvtph\_pd (\_\_m128h a);  
 VCVTPH2PD \_\_m128d \_\_mm\_mask\_cvtph\_pd (\_\_m128d src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2PD \_\_m128d \_\_mm\_maskz\_cvtph\_pd (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2PD \_\_m256d \_\_mm256\_cvtph\_pd (\_\_m128h a);  
 VCVTPH2PD \_\_m256d \_\_mm256\_mask\_cvtph\_pd (\_\_m256d src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2PD \_\_m256d \_\_mm256\_maskz\_cvtph\_pd (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2PD \_\_m512d \_\_mm512\_cvtph\_pd (\_\_m128h a);  
 VCVTPH2PD \_\_m512d \_\_mm512\_mask\_cvtph\_pd (\_\_m512d src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2PD \_\_m512d \_\_mm512\_maskz\_cvtph\_pd (\_\_mmask8 k, \_\_m128h a);

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

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

## VCVTPH2PS/VCVTPH2PSX—Convert Packed FP16 Values to Single Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature Flag   | Description   |
|---|------------|------------------------------|-------------------------|---|
| VEX.128.66.0F38.W0 13 /r<br>VCVTPH2PS xmm1, xmm2/m64                            | A          | V/V                          | F16C                    | Convert four packed FP16 values in xmm2/m64 to packed single precision floating-point value in xmm1.  |
| VEX.256.66.0F38.W0 13 /r<br>VCVTPH2PS ymm1, xmm2/m128                           | A          | V/V                          | F16C                    | Convert eight packed FP16 values in xmm2/m128 to packed single precision floating-point value in ymm1.  |
| EVEX.128.66.0F38.W0 13 /r<br>VCVTPH2PS xmm1 {k1}{z},<br>xmm2/m64                | B          | V/V                          | AVX512VL<br>AVX512F     | Convert four packed FP16 values in xmm2/m64 to packed single precision floating-point values in xmm1 subject to writemask k1.                                       |
| EVEX.256.66.0F38.W0 13 /r<br>VCVTPH2PS ymm1 {k1}{z},<br>xmm2/m128               | B          | V/V                          | AVX512VL<br>AVX512F     | Convert eight packed FP16 values in xmm2/m128 to packed single precision floating-point values in ymm1 subject to writemask k1.                                     |
| EVEX.512.66.0F38.W0 13 /r<br>VCVTPH2PS zmm1 {k1}{z},<br>ymm2/m256 {sae}         | B          | V/V                          | AVX512F                 | Convert sixteen packed FP16 values in ymm2/m256 to packed single precision floating-point values in zmm1 subject to writemask k1.                                   |
| EVEX.128.66.MAP6.W0 13 /r<br>VCVTPH2PSX xmm1{k1}{z},<br>xmm2/m64/m16bcst        | C          | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed FP16 values in xmm2/m64/m16bcst to four packed single precision floating-point values, and store result in xmm1 subject to writemask k1.        |
| EVEX.256.66.MAP6.W0 13 /r<br>VCVTPH2PSX ymm1{k1}{z},<br>xmm2/m128/m16bcst       | C          | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed FP16 values in xmm2/m128/m16bcst to eight packed single precision floating-point values, and store result in ymm1 subject to writemask k1.     |
| EVEX.512.66.MAP6.W0 13 /r<br>VCVTPH2PSX zmm1{k1}{z},<br>ymm2/m256/m16bcst {sae} | C          | V/V                          | AVX512-FP16             | Convert sixteen packed FP16 values in ymm2/m256/m16bcst to sixteen packed single precision floating-point values, and store result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | N/A        | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| B     | Half Mem   | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| C     | Half       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction 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) floating-point values.

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

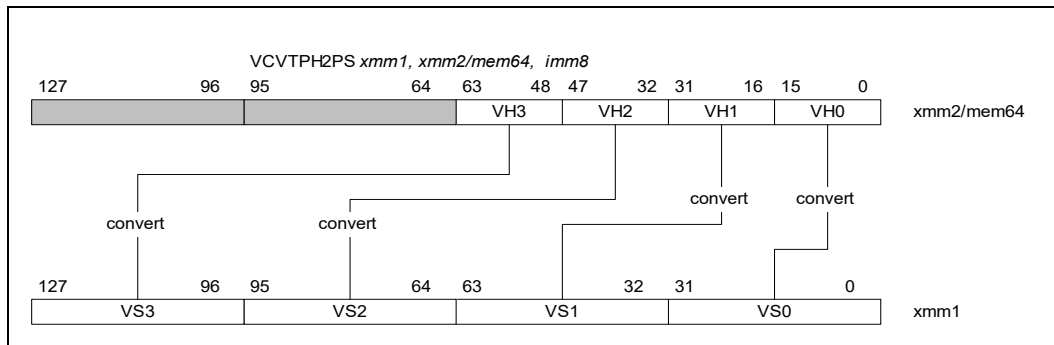


Figure 5-6. VCVTPH2PS (128-bit Version)

The VCVTPH2PSX instruction is a new form of the PH to PS conversion instruction, encoded in map 6. The previous version of the instruction, VCVTPH2PS, that is present in AVX512F (encoded in map 2, 0F38) does not support embedded broadcasting. The VCVTPH2PSX instruction has the embedded broadcasting option available.

The instructions associated with AVX512\_FP16 always handle FP16 denormal number inputs; denormal inputs are not treated as zero.

**Operation**

```
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 (VEX.128 Encoded Version)**

```

DEST[31:0] := vCvt_h2s(SRC1[15:0]);
DEST[63:32] := vCvt_h2s(SRC1[31:16]);
DEST[95:64] := vCvt_h2s(SRC1[47:32]);
DEST[127:96] := vCvt_h2s(SRC1[63:48]);
DEST[MAXVL-1:128] := 0

```

**VCVTPH2PSX DEST, SRC**

VL = 128, 256, or 512

KL := VL/32

FOR j := 0 TO KL-1:

```

    IF k1[j] OR *no writemask*:
        IF *SRC is memory* and EVEX.b = 1:
            tsrc := SRC.fp16[0]
        ELSE
            tsrc := SRC.fp16[j]
        DEST.fp32[j] := Convert_fp16_to_fp32(tsrc)
    ELSE IF *zeroing*:
        DEST.fp32[j] := 0
    // else dest.fp32[j] remains unchanged

```

DEST[MAXVL-1:VL] := 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_mask_cvtph_ps(__m256 s, __mmask8 k, __m128i a);
VCVTPH2PS __m256 __mm256_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 __mm_mask_cvtph_ps(__m128 s, __mmask8 k, __m128i a);
VCVTPH2PS __m128 __mm_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 __mm_cvtph_ps ( __m128i m1);
VCVTPH2PS __m256 __mm256_cvtph_ps ( __m128i m1)

VCVTPH2PSX __m512 __mm512_cvtx_roundph_ps ( __m256h a, int sae);
VCVTPH2PSX __m512 __mm512_mask_cvtx_roundph_ps ( __m512 src, __mmask16 k, __m256h a, int sae);

```

VCVTPH2PSX \_\_m512 \_\_mm512\_maskz\_cvtx\_roundph\_ps (\_\_mmask16 k, \_\_m256h a, int sae);  
 VCVTPH2PSX \_\_m128 \_\_mm\_cvtxph\_ps (\_\_m128h a);  
 VCVTPH2PSX \_\_m128 \_\_mm\_mask\_cvtxph\_ps (\_\_m128 src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2PSX \_\_m128 \_\_mm\_maskz\_cvtxph\_ps (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2PSX \_\_m256 \_\_mm256\_cvtxph\_ps (\_\_m128h a);  
 VCVTPH2PSX \_\_m256 \_\_mm256\_mask\_cvtxph\_ps (\_\_m256 src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2PSX \_\_m256 \_\_mm256\_maskz\_cvtxph\_ps (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2PSX \_\_m512 \_\_mm512\_cvtxph\_ps (\_\_m256h a);  
 VCVTPH2PSX \_\_m512 \_\_mm512\_mask\_cvtxph\_ps (\_\_m512 src, \_\_mmask16 k, \_\_m256h a);  
 VCVTPH2PSX \_\_m512 \_\_mm512\_maskz\_cvtxph\_ps (\_\_mmask16 k, \_\_m256h a);

### SIMD Floating-Point Exceptions

VEX-encoded instructions: Invalid.

EVEX-encoded instructions: Invalid.

EVEX-encoded instructions with broadcast (VCVTPH2PSX): Invalid, Denormal.

### 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.”

EVEX-encoded instructions with broadcast (VCVTPH2PSX), see Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD If VEX.W=1.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VCVTPH2QQ—Convert Packed FP16 Values to Signed Quadword Integer Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|-----------|------------------------------|-------------------------|--|
| EVEX.128.66.MAP5.W0 7B /r<br>VCVTPH2QQ xmm1{k1}{z}, xmm2/<br>m32/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert two packed FP16 values in xmm2/m32/<br>m16bcst to two signed quadword integers, and<br>store the result in xmm1 subject to writemask k1.         |
| EVEX.256.66.MAP5.W0 7B /r<br>VCVTPH2QQ ymm1{k1}{z}, xmm2/<br>m64/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed FP16 values in xmm2/m64/<br>m16bcst to four signed quadword integers, and<br>store the result in ymm1 subject to writemask k1.       |
| EVEX.512.66.MAP5.W0 7B /r<br>VCVTPH2QQ zmm1{k1}{z}, xmm2/<br>m128/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight signed quadword<br>integers, and store the result in zmm1 subject to<br>writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------|---------------|---------------|-----------|-----------|
| A     | Quarter | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to signed quadword integers in 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 is returned.

The destination elements are updated according to the writemask.

### Operation

#### VCVTPH2QQ DEST, SRC

VL = 128, 256 or 512

KL := VL / 64

IF \*SRC is a register\* and (VL = 512) and (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.fp16[0]

ELSE

tsrc := SRC.fp16[j]

DEST.qword[j] := Convert\_fp16\_to\_integer64(tsrc)

ELSE IF \*zeroing\*:

DEST.qword[j] := 0

// else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTPH2QQ \_\_m512i \_\_mm512\_cvt\_roundph\_epi64 (\_\_m128h a, int rounding);  
 VCVTPH2QQ \_\_m512i \_\_mm512\_mask\_cvt\_roundph\_epi64 (\_\_m512i src, \_\_mmask8 k, \_\_m128h a, int rounding);  
 VCVTPH2QQ \_\_m512i \_\_mm512\_maskz\_cvt\_roundph\_epi64 (\_\_mmask8 k, \_\_m128h a, int rounding);  
 VCVTPH2QQ \_\_m128i \_\_mm\_cvtph\_epi64 (\_\_m128h a);  
 VCVTPH2QQ \_\_m128i \_\_mm\_mask\_cvtph\_epi64 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2QQ \_\_m128i \_\_mm\_maskz\_cvtph\_epi64 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2QQ \_\_m256i \_\_mm256\_cvtph\_epi64 (\_\_m128h a);  
 VCVTPH2QQ \_\_m256i \_\_mm256\_mask\_cvtph\_epi64 (\_\_m256i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2QQ \_\_m256i \_\_mm256\_maskz\_cvtph\_epi64 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2QQ \_\_m512i \_\_mm512\_cvtph\_epi64 (\_\_m128h a);  
 VCVTPH2QQ \_\_m512i \_\_mm512\_mask\_cvtph\_epi64 (\_\_m512i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2QQ \_\_m512i \_\_mm512\_maskz\_cvtph\_epi64 (\_\_mmask8 k, \_\_m128h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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



## VCVTPH2UDQ—Convert Packed FP16 Values to Unsigned Doubleword Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.NP.MAP5.W0 79 /r<br>VCVTPH2UDQ xmm1{k1}{z}, xmm2/<br>m64/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed FP16 values in xmm2/m64/<br>m16bcst to four unsigned doubleword integers,<br>and store the result in xmm1 subject to<br>writemask k1.        |
| EVEX.256.NP.MAP5.W0 79 /r<br>VCVTPH2UDQ ymm1{k1}{z}, xmm2/<br>m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight unsigned doubleword<br>integers, and store the result in ymm1 subject to<br>writemask k1.     |
| EVEX.512.NP.MAP5.W0 79 /r<br>VCVTPH2UDQ zmm1{k1}{z}, ymm2/<br>m256/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Convert sixteen packed FP16 values in ymm2/<br>m256/m16bcst to sixteen unsigned doubleword<br>integers, and store the result in zmm1 subject to<br>writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Half  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to unsigned doubleword integers in 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 is returned.

The destination elements are updated according to the writemask.

### Operation

#### VCVTPH2UDQ DEST, SRC

VL = 128, 256 or 512

KL := VL / 32

IF \*SRC is a register\* and (VL = 512) and (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.fp16[0]

ELSE

tsrc := SRC.fp16[j]

DEST.dword[j] := Convert\_fp16\_to\_unsigned\_integer32(tsrc)

ELSE IF \*zeroing\*:

DEST.dword[j] := 0

// else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTPH2UDQ \_\_m512i \_mm512\_cvt\_roundph\_epu32 (\_\_m256h a, int rounding);  
 VCVTPH2UDQ \_\_m512i \_mm512\_mask\_cvt\_roundph\_epu32 (\_\_m512i src, \_\_mmask16 k, \_\_m256h a, int rounding);  
 VCVTPH2UDQ \_\_m512i \_mm512\_maskz\_cvt\_roundph\_epu32 (\_\_mmask16 k, \_\_m256h a, int rounding);  
 VCVTPH2UDQ \_\_m128i \_mm\_cvtph\_epu32 (\_\_m128h a);  
 VCVTPH2UDQ \_\_m128i \_mm\_mask\_cvtph\_epu32 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UDQ \_\_m128i \_mm\_maskz\_cvtph\_epu32 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UDQ \_\_m256i \_mm256\_cvtph\_epu32 (\_\_m128h a);  
 VCVTPH2UDQ \_\_m256i \_mm256\_mask\_cvtph\_epu32 (\_\_m256i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UDQ \_\_m256i \_mm256\_maskz\_cvtph\_epu32 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UDQ \_\_m512i \_mm512\_cvtph\_epu32 (\_\_m256h a);  
 VCVTPH2UDQ \_\_m512i \_mm512\_mask\_cvtph\_epu32 (\_\_m512i src, \_\_mmask16 k, \_\_m256h a);  
 VCVTPH2UDQ \_\_m512i \_mm512\_maskz\_cvtph\_epu32 (\_\_mmask16 k, \_\_m256h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

## VCVTPH2UQQ—Convert Packed FP16 Values to Unsigned Quadword Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.66.MAP5.W0 79 /r<br>VCVTPH2UQQ xmm1{k1}{z}, xmm2/<br>m32/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert two packed FP16 values in xmm2/m32/<br>m16bcst to two unsigned quadword integers, and<br>store the result in xmm1 subject to writemask k1.         |
| EVEX.256.66.MAP5.W0 79 /r<br>VCVTPH2UQQ ymm1{k1}{z}, xmm2/<br>m64/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed FP16 values in xmm2/m64/<br>m16bcst to four unsigned quadword integers,<br>and store the result in ymm1 subject to<br>writemask k1.    |
| EVEX.512.66.MAP5.W0 79 /r<br>VCVTPH2UQQ zmm1{k1}{z}, xmm2/<br>m128/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight unsigned quadword<br>integers, and store the result in zmm1 subject to<br>writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------|---------------|---------------|-----------|-----------|
| A     | Quarter | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to unsigned quadword integers in 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 is returned.

The destination elements are updated according to the writemask.

### Operation

#### VCVTPH2UQQ DEST, SRC

VL = 128, 256 or 512

KL := VL / 64

IF \*SRC is a register\* and (VL = 512) and (EVEX.b = 1):

    SET\_RM(EVEX.RC)

ELSE:

    SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

    IF k1[j] OR \*no writemask\*:

        IF \*SRC is memory\* and EVEX.b = 1:

            tsrc := SRC.fp16[0]

        ELSE

            tsrc := SRC.fp16[j]

        DEST.qword[j] := Convert\_fp16\_to\_unsigned\_integer64(tsrc)

    ELSE IF \*zeroing\*:

        DEST.qword[j] := 0

    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTPH2UQQ \_\_m512i \_mm512\_cvt\_roundph\_epu64 (\_\_m128h a, int rounding);  
 VCVTPH2UQQ \_\_m512i \_mm512\_mask\_cvt\_roundph\_epu64 (\_\_m512i src, \_\_mmask8 k, \_\_m128h a, int rounding);  
 VCVTPH2UQQ \_\_m512i \_mm512\_maskz\_cvt\_roundph\_epu64 (\_\_mmask8 k, \_\_m128h a, int rounding);  
 VCVTPH2UQQ \_\_m128i \_mm\_cvtph\_epu64 (\_\_m128h a);  
 VCVTPH2UQQ \_\_m128i \_mm\_mask\_cvtph\_epu64 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UQQ \_\_m128i \_mm\_maskz\_cvtph\_epu64 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UQQ \_\_m256i \_mm256\_cvtph\_epu64 (\_\_m128h a);  
 VCVTPH2UQQ \_\_m256i \_mm256\_mask\_cvtph\_epu64 (\_\_m256i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UQQ \_\_m256i \_mm256\_maskz\_cvtph\_epu64 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UQQ \_\_m512i \_mm512\_cvtph\_epu64 (\_\_m128h a);  
 VCVTPH2UQQ \_\_m512i \_mm512\_mask\_cvtph\_epu64 (\_\_m512i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UQQ \_\_m512i \_mm512\_maskz\_cvtph\_epu64 (\_\_mmask8 k, \_\_m128h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

## VCVTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.MAP5.WO 7D /r<br>VCVTPH2UW xmm1{k1}{z}, xmm2/<br>m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert packed FP16 values in xmm2/m128/<br>m16bcst to unsigned word integers, and store the<br>result in xmm1. |
| EVEX.256.NP.MAP5.WO 7D /r<br>VCVTPH2UW ymm1{k1}{z}, ymm2/<br>m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert packed FP16 values in ymm2/m256/<br>m16bcst to unsigned word integers, and store the<br>result in ymm1. |
| EVEX.512.NP.MAP5.WO 7D /r<br>VCVTPH2UW zmm1{k1}{z}, zmm2/<br>m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Convert packed FP16 values in zmm2/m512/<br>m16bcst to unsigned word integers, and store the<br>result in zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to unsigned 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 indefinite integer value is returned.

The destination elements are updated according to the writemask.

### Operation

#### VCVTPH2UW DEST, SRC

VL = 128, 256 or 512

KL := VL / 16

IF \*SRC is a register\* and (VL = 512) and (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.fp16[0]

ELSE

tsrc := SRC.fp16[j]

DEST.word[j] := Convert\_fp16\_to\_unsigned\_integer16(tsrc)

ELSE IF \*zeroing\*:

DEST.word[j] := 0

// else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTPH2UW \_\_m512i \_\_mm512\_cvt\_roundph\_epu16 (\_\_m512h a, int sae);  
 VCVTPH2UW \_\_m512i \_\_mm512\_mask\_cvt\_roundph\_epu16 (\_\_m512i src, \_\_mmask32 k, \_\_m512h a, int sae);  
 VCVTPH2UW \_\_m512i \_\_mm512\_maskz\_cvt\_roundph\_epu16 (\_\_mmask32 k, \_\_m512h a, int sae);  
 VCVTPH2UW \_\_m128i \_\_mm\_cvtph\_epu16 (\_\_m128h a);  
 VCVTPH2UW \_\_m128i \_\_mm\_mask\_cvtph\_epu16 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UW \_\_m128i \_\_mm\_maskz\_cvtph\_epu16 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2UW \_\_m256i \_\_mm256\_cvtph\_epu16 (\_\_m256h a);  
 VCVTPH2UW \_\_m256i \_\_mm256\_mask\_cvtph\_epu16 (\_\_m256i src, \_\_mmask16 k, \_\_m256h a);  
 VCVTPH2UW \_\_m256i \_\_mm256\_maskz\_cvtph\_epu16 (\_\_mmask16 k, \_\_m256h a);  
 VCVTPH2UW \_\_m512i \_\_mm512\_cvtph\_epu16 (\_\_m512h a);  
 VCVTPH2UW \_\_m512i \_\_mm512\_mask\_cvtph\_epu16 (\_\_m512i src, \_\_mmask32 k, \_\_m512h a);  
 VCVTPH2UW \_\_m512i \_\_mm512\_maskz\_cvtph\_epu16 (\_\_mmask32 k, \_\_m512h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

## VCVTPH2W—Convert Packed FP16 Values to Signed Word Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP5.W0 7D /r<br>VCVTPH2W xmm1{k1}{z}, xmm2/<br>m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert packed FP16 values in xmm2/m128/<br>m16bcst to signed word integers, and store the<br>result in xmm1. |
| EVEX.256.66.MAP5.W0 7D /r<br>VCVTPH2W ymm1{k1}{z}, ymm2/<br>m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert packed FP16 values in ymm2/m256/<br>m16bcst to signed word integers, and store the<br>result in ymm1. |
| EVEX.512.66.MAP5.W0 7D /r<br>VCVTPH2W zmm1{k1}{z}, zmm2/<br>m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Convert packed FP16 values in zmm2/m512/<br>m16bcst to signed word integers, and store the<br>result in zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to signed 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 indefinite integer value is returned.

The destination elements are updated according to the writemask.

### Operation

#### VCVTPH2W DEST, SRC

VL = 128, 256 or 512

KL := VL / 16

IF \*SRC is a register\* and (VL = 512) and (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.fp16[0]

ELSE

tsrc := SRC.fp16[j]

DEST.word[j] := Convert\_fp16\_to\_integer16(tsrc)

ELSE IF \*zeroing\*:

DEST.word[j] := 0

// else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTPH2W \_\_m512i \_\_mm512\_cvt\_roundph\_epi16 (\_\_m512h a, int rounding);  
 VCVTPH2W \_\_m512i \_\_mm512\_mask\_cvt\_roundph\_epi16 (\_\_m512i src, \_\_mmask32 k, \_\_m512h a, int rounding);  
 VCVTPH2W \_\_m512i \_\_mm512\_maskz\_cvt\_roundph\_epi16 (\_\_mmask32 k, \_\_m512h a, int rounding);  
 VCVTPH2W \_\_m128i \_\_mm\_cvtph\_epi16 (\_\_m128h a);  
 VCVTPH2W \_\_m128i \_\_mm\_mask\_cvtph\_epi16 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTPH2W \_\_m128i \_\_mm\_maskz\_cvtph\_epi16 (\_\_mmask8 k, \_\_m128h a);  
 VCVTPH2W \_\_m256i \_\_mm256\_cvtph\_epi16 (\_\_m256h a);  
 VCVTPH2W \_\_m256i \_\_mm256\_mask\_cvtph\_epi16 (\_\_m256i src, \_\_mmask16 k, \_\_m256h a);  
 VCVTPH2W \_\_m256i \_\_mm256\_maskz\_cvtph\_epi16 (\_\_mmask16 k, \_\_m256h a);  
 VCVTPH2W \_\_m512i \_\_mm512\_cvtph\_epi16 (\_\_m512h a);  
 VCVTPH2W \_\_m512i \_\_mm512\_mask\_cvtph\_epi16 (\_\_m512i src, \_\_mmask32 k, \_\_m512h a);  
 VCVTPH2W \_\_m512i \_\_mm512\_maskz\_cvtph\_epi16 (\_\_mmask32 k, \_\_m512h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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



## VCVTSP2PH—Convert Single-Precision FP Value to 16-bit FP Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| VEX.128.66.0F3A.W0 1D /r ib<br>VCVTSP2PH xmm1/m64, xmm2,<br>imm8                | A         | V/V                          | F16C                     | 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.     |
| VEX.256.66.0F3A.W0 1D /r ib<br>VCVTSP2PH xmm1/m128, ymm2,<br>imm8               | A         | V/V                          | F16C                     | 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.   |
| EVEX.128.66.0F3A.W0 1D /r ib<br>VCVTSP2PH xmm1/m64 {k1}{z},<br>xmm2, imm8       | B         | V/V                          | AVX512VL<br>AVX512F      | 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.     |
| EVEX.256.66.0F3A.W0 1D /r ib<br>VCVTSP2PH xmm1/m128 {k1}{z},<br>ymm2, imm8      | B         | V/V                          | AVX512VL<br>AVX512F      | 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.   |
| EVEX.512.66.0F3A.W0 1D /r ib<br>VCVTSP2PH ymm1/m256 {k1}{z},<br>zmm2{sae}, imm8 | B         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | N/A        | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |
| B     | Half Mem   | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |

### 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.

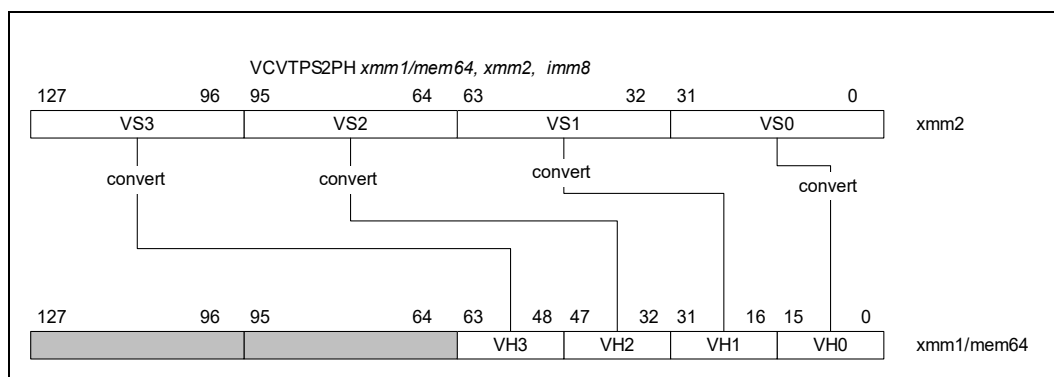


Figure 5-7. VCVTSP2PH (128-bit Version)

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

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

| Bits     | Field Name/value | Description               | Comment         |
|----------|------------------|---------------------------|-----------------|
| Imm[1:0] | RC=00B           | Round to nearest even     | If Imm[2] = 0   |
|          | RC=01B           | Round down                |                 |
|          | RC=10B           | Round up                  |                 |
|          | RC=11B           | Truncate                  |                 |
| Imm[2]   | MS1=0            | Use imm[1:0] for rounding | Ignore MXCSR.RC |
|          | MS1=1            | Use MXCSR.RC for rounding |                 |
| Imm[7:3] | Ignored          | Ignored by processor      |                 |

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

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

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

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

### Operation

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

**VCVTPS2PH (EVEX Encoded Versions) When DEST is Memory**

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

```

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

```

**VCVTPS2PH (VEX.256 Encoded Version)**

```

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

```

**VCVTPS2PH (VEX.128 Encoded Version)**

```

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

```

**Flags Affected**

None.

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

```

VCVTPS2PH __m256i __mm512_cvtps_ph(__m512 a);
VCVTPS2PH __m256i __mm512_mask_cvtps_ph(__m256i s, __mmask16 k, __m512 a);
VCVTPS2PH __m256i __mm512_maskz_cvtps_ph(__mmask16 k, __m512 a);
VCVTPS2PH __m256i __mm512_cvt_roundps_ph(__m512 a, const int imm);
VCVTPS2PH __m256i __mm512_mask_cvt_roundps_ph(__m256i s, __mmask16 k, __m512 a, const int imm);
VCVTPS2PH __m256i __mm512_maskz_cvt_roundps_ph(__mmask16 k, __m512 a, const int imm);
VCVTPS2PH __m128i __mm256_mask_cvtps_ph(__m128i s, __mmask8 k, __m256 a);
VCVTPS2PH __m128i __mm256_maskz_cvtps_ph(__mmask8 k, __m256 a);
VCVTPS2PH __m128i __mm_mask_cvtps_ph(__m128i s, __mmask8 k, __m128 a);
VCVTPS2PH __m128i __mm_maskz_cvtps_ph(__mmask8 k, __m128 a);
VCVTPS2PH __m128i __mm_cvtps_ph (__m128 m1, const int imm);
VCVTPS2PH __m128i __mm256_cvtps_ph(__m256 m1, const int imm);

```

**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.vvvv != 1111B or EVEX.vvvv != 1111B.

## VCVTPS2PHX—Convert Packed Single Precision Floating-Point Values to Packed FP16 Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|------------|------------------------------|-------------------------|--|
| EVEX.128.66.MAP5.W0 1D /r<br>VCVTPS2PHX xmm1{k1}{z},<br>xmm2/m128/m32bcst      | A          | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed single precision floating-point values in xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.     |
| EVEX.256.66.MAP5.W0 1D /r<br>VCVTPS2PHX ymm1{k1}{z},<br>ymm2/m256/m32bcst      | A          | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed single precision floating-point values in ymm2/m256/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1.    |
| EVEX.512.66.MAP5.W0 1D /r<br>VCVTPS2PHX zmm1{k1}{z},<br>zmm2/m512/m32bcst {er} | A          | V/V                          | AVX512-FP16             | Convert sixteen packed single precision floating-point values in zmm2 /m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed single precision floating values in the source operand to FP16 values and stores to the destination operand.

The VCVTPS2PHX instruction supports broadcasting.

This instruction uses MXCSR.DAZ for handling FP32 inputs. FP16 outputs can be normal or denormal numbers, and are not conditionally flushed based on MXCSR settings.

### Operation

#### VCVTPS2PHX DEST, SRC (AVX512\_FP16 Load Version With Broadcast Support)

VL = 128, 256, or 512

KL := VL / 32

IF \*SRC is a register\* and (VL == 512) and (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.fp32[0]

ELSE

tsrc := SRC.fp32[j]

DEST.fp16[j] := Convert\_fp32\_to\_fp16(tsrc)

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

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

**Flags Affected**

None.

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

VCVTSP2PHX \_\_m256h \_\_mm512\_cvtx\_roundps\_ph (\_\_m512 a, int rounding);  
 VCVTSP2PHX \_\_m256h \_\_mm512\_mask\_cvtx\_roundps\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m512 a, int rounding);  
 VCVTSP2PHX \_\_m256h \_\_mm512\_maskz\_cvtx\_roundps\_ph (\_\_mmask16 k, \_\_m512 a, int rounding);  
 VCVTSP2PHX \_\_m128h \_\_mm\_cvtxps\_ph (\_\_m128 a);  
 VCVTSP2PHX \_\_m128h \_\_mm\_mask\_cvtxps\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128 a);  
 VCVTSP2PHX \_\_m128h \_\_mm\_maskz\_cvtxps\_ph (\_\_mmask8 k, \_\_m128 a);  
 VCVTSP2PHX \_\_m128h \_\_mm256\_cvtxps\_ph (\_\_m256 a);  
 VCVTSP2PHX \_\_m128h \_\_mm256\_mask\_cvtxps\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m256 a);  
 VCVTSP2PHX \_\_m128h \_\_mm256\_maskz\_cvtxps\_ph (\_\_mmask8 k, \_\_m256 a);  
 VCVTSP2PHX \_\_m256h \_\_mm512\_cvtxps\_ph (\_\_m512 a);  
 VCVTSP2PHX \_\_m256h \_\_mm512\_mask\_cvtxps\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m512 a);  
 VCVTSP2PHX \_\_m256h \_\_mm512\_maskz\_cvtxps\_ph (\_\_mmask16 k, \_\_m512 a);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0).

**Other Exceptions**

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

Additionally:

#UD                    If VEX.W=1.  
 #UD                    If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VCVTQPS2Q—Convert Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.66.0F.W0 7B /r<br>VCVTQPS2Q xmm1 {k1}{z},<br>xmm2/m64/m32bcst      | A          | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.      |
| EVEX.256.66.0F.W0 7B /r<br>VCVTQPS2Q ymm1 {k1}{z},<br>xmm2/m128/m32bcst     | A          | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.   |
| EVEX.512.66.0F.W0 7B /r<br>VCVTQPS2Q zmm1 {k1}{z},<br>ymm2/m256/m32bcst{er} | A          | V/V                          | AVX512DQ                 | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Half       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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

#### VCVTQPS2Q (EVEX Encoded Versions) When SRC Operand is a Register

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 64

k := j \* 32

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

THEN DEST[i+63:i] :=

Convert\_Single\_Precision\_To\_QuadInteger(SRC[k+31:k])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

```

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

```

### VCVTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Memory Source (KL, VL) = (2, 128), (4, 256), (8, 512)

```

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

```

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

```

VCVTPS2QQ __m512i __mm512_cvtps_epi64( __m512 a);
VCVTPS2QQ __m512i __mm512_mask_cvtps_epi64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i __mm512_maskz_cvtps_epi64( __mmask16 k, __m512 a);
VCVTPS2QQ __m512i __mm512_cvt_roundps_epi64( __m512 a, int r);
VCVTPS2QQ __m512i __mm512_mask_cvt_roundps_epi64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i __mm512_maskz_cvt_roundps_epi64( __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i __mm256_cvtps_epi64( __m256 a);
VCVTPS2QQ __m256i __mm256_mask_cvtps_epi64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i __mm256_maskz_cvtps_epi64( __mmask8 k, __m256 a);
VCVTPS2QQ __m128i __mm_cvtps_epi64( __m128 a);
VCVTPS2QQ __m128i __mm_mask_cvtps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i __mm_maskz_cvtps_epi64( __mmask8 k, __m128 a);

```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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

Additionally:

#UD                    If EVEX.vvvv != 1111B.



## VCVTPS2UDQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

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

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

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

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

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

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

**Operation****VCVTPS2UDQ (EVEX Encoded Versions) When SRC Operand is a Register**

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

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

```

    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;

FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI;
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 __mm512_cvtps_epu32( __m512 a);
VCVTPS2UDQ __m512i __mm512_mask_cvtps_epu32( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i __mm512_maskz_cvtps_epu32( __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i __mm512_cvt_roundps_epu32( __m512 a, int r);
VCVTPS2UDQ __m512i __mm512_mask_cvt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i __mm512_maskz_cvt_roundps_epu32( __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i __mm256_cvtps_epu32( __m256d a);
VCVTPS2UDQ __m256i __mm256_mask_cvtps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i __mm256_maskz_cvtps_epu32( __mmask8 k, __m256 a);
VCVTPS2UDQ __m128i __mm_cvtps_epu32( __m128 a);
VCVTPS2UDQ __m128i __mm_mask_cvtps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i __mm_maskz_cvtps_epu32( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

Additionally:

#UD                    If EVEX.vvvv != 1111B.

## VCVTQPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F.W0 79 /r<br>VCVTQPS2UQQ xmm1 {k1}{z},<br>xmm2/m64/m32bcst      | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed single precision floating-point values from zmm2/m64/m32bcst to two packed unsigned quadword values in zmm1 subject to writemask k1.      |
| EVEX.256.66.0F.W0 79 /r<br>VCVTQPS2UQQ ymm1 {k1}{z},<br>xmm2/m128/m32bcst     | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.   |
| EVEX.512.66.0F.W0 79 /r<br>VCVTQPS2UQQ zmm1 {k1}{z},<br>ymm2/m256/m32bcst{er} | A         | V/V                          | AVX512DQ                 | Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Half       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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

#### VCVTQPS2UQQ (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;
  ENDFOR
  DEST[MAXVL-1:VL] := 0

```

### VCVTSP2UQQ (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;
      ENDFOR
      DEST[MAXVL-1:VL] := 0

```

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

```

VCVTSP2UQQ __m512i __mm512_cvtps_epu64( __m512 a);
VCVTSP2UQQ __m512i __mm512_mask_cvtps_epu64( __m512i s, __mmask16 k, __m512 a);
VCVTSP2UQQ __m512i __mm512_maskz_cvtps_epu64( __mmask16 k, __m512 a);
VCVTSP2UQQ __m512i __mm512_cvt_roundps_epu64( __m512 a, int r);
VCVTSP2UQQ __m512i __mm512_mask_cvt_roundps_epu64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTSP2UQQ __m512i __mm512_maskz_cvt_roundps_epu64( __mmask16 k, __m512 a, int r);
VCVTSP2UQQ __m256i __mm256_cvtps_epu64( __m256 a);
VCVTSP2UQQ __m256i __mm256_mask_cvtps_epu64( __m256i s, __mmask8 k, __m256 a);
VCVTSP2UQQ __m256i __mm256_maskz_cvtps_epu64( __mmask8 k, __m256 a);
VCVTSP2UQQ __m128i __mm_cvtps_epu64( __m128 a);
VCVTSP2UQQ __m128i __mm_mask_cvtps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTSP2UQQ __m128i __mm_maskz_cvtps_epu64( __mmask8 k, __m128 a);

```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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

Additionally:

#UD                    If EVEX.vvvv != 1111B.

## VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F.W1 E6 /r<br>VCVTQQ2PD xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed quadword integers from xmm2/m128/m64bcst to packed double precision floating-point values in xmm1 with writemask k1.         |
| EVEX.256.F3.0F.W1 E6 /r<br>VCVTQQ2PD ymm1 {k1}{z},<br>ymm2/m256/m64bcst     | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed quadword integers from ymm2/m256/m64bcst to packed double precision floating-point values in ymm1 with writemask k1.        |
| EVEX.512.F3.0F.W1 E6 /r<br>VCVTQQ2PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | A         | V/V                          | AVX512DQ                 | Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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;

ENDFOR

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, int r);

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.

## VCVTQQ2PH—Convert Packed Signed Quadword Integers to Packed FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.MAP5.W1 5B /r<br>VCVTQQ2PH xmm1{k1}{z}, xmm2/<br>m128/m64bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert two packed signed quadword integers in xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.   |
| EVEX.256.NP.MAP5.W1 5B /r<br>VCVTQQ2PH xmm1{k1}{z}, ymm2/<br>m256/m64bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed signed quadword integers in ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.  |
| EVEX.512.NP.MAP5.W1 5B /r<br>VCVTQQ2PH xmm1{k1}{z}, zmm2/<br>m512/m64bcst {er} | A         | V/V                          | AVX512-FP16             | Convert eight packed signed quadword integers in zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

#### Description

This instruction converts packed signed quadword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

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

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

#### Operation

##### VCVTQQ2PH DEST, SRC

VL = 128, 256 or 512

KL := VL / 64

IF \*SRC is a register\* and (VL = 512) AND (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.qword[0]

ELSE

tsrc := SRC.qword[j]

DEST.fp16[j] := Convert\_integer64\_to\_fp16(tsrc)

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0



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

VCVTQQ2PH \_\_m128h \_\_mm512\_cvt\_roundepi64\_ph (\_\_m512i a, int rounding);  
 VCVTQQ2PH \_\_m128h \_\_mm512\_mask\_cvt\_roundepi64\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m512i a, int rounding);  
 VCVTQQ2PH \_\_m128h \_\_mm512\_maskz\_cvt\_roundepi64\_ph (\_\_mmask8 k, \_\_m512i a, int rounding);  
 VCVTQQ2PH \_\_m128h \_\_mm\_cvtepi64\_ph (\_\_m128i a);  
 VCVTQQ2PH \_\_m128h \_\_mm\_mask\_cvtepi64\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128i a);  
 VCVTQQ2PH \_\_m128h \_\_mm\_maskz\_cvtepi64\_ph (\_\_mmask8 k, \_\_m128i a);  
 VCVTQQ2PH \_\_m128h \_\_mm256\_cvtepi64\_ph (\_\_m256i a);  
 VCVTQQ2PH \_\_m128h \_\_mm256\_mask\_cvtepi64\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m256i a);  
 VCVTQQ2PH \_\_m128h \_\_mm256\_maskz\_cvtepi64\_ph (\_\_mmask8 k, \_\_m256i a);  
 VCVTQQ2PH \_\_m128h \_\_mm512\_cvtepi64\_ph (\_\_m512i a);  
 VCVTQQ2PH \_\_m128h \_\_mm512\_mask\_cvtepi64\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m512i a);  
 VCVTQQ2PH \_\_m128h \_\_mm512\_maskz\_cvtepi64\_ph (\_\_mmask8 k, \_\_m512i a);

**SIMD Floating-Point Exceptions**

Overflow, Precision.

**Other Exceptions**

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

## VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.0F.W1 5B /r<br>VCVTQQ2PS xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed quadword integers from xmm2/mem to packed single precision floating-point values in xmm1 with writemask k1.         |
| EVEX.256.0F.W1 5B /r<br>VCVTQQ2PS xmm1 {k1}{z},<br>ymm2/m256/m64bcst     | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed quadword integers from ymm2/mem to packed single precision floating-point values in xmm1 with writemask k1.        |
| EVEX.512.0F.W1 5B /r<br>VCVTQQ2PS ymm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | A         | V/V                          | AVX512DQ                 | Convert eight packed quadword integers from zmm2/mem to eight packed single precision floating-point values in ymm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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[j+63:i])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[k+31:k] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[k+31:k] := 0
  FI
FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0

```

**VCVTQQ2PS (EVEX Encoded Versions) When SRC Operand is a Memory Source**

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

```

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

```

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

```

VCVTQQ2PS __m256 __mm512_cvtepi64_ps( __m512i a);
VCVTQQ2PS __m256 __mm512_mask_cvtepi64_ps( __m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 __mm512_maskz_cvtepi64_ps( __mmask16 k, __m512i a);
VCVTQQ2PS __m256 __mm512_cvt_roundepi64_ps( __m512i a, int r);
VCVTQQ2PS __m256 __mm512_mask_cvt_roundepi64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m256 __mm512_maskz_cvt_roundepi64_ps( __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m128 __mm256_cvtepi64_ps( __m256i a);
VCVTQQ2PS __m128 __mm256_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTQQ2PS __m128 __mm256_maskz_cvtepi64_ps( __mmask8 k, __m256i a);
VCVTQQ2PS __m128 __mm_cvtepi64_ps( __m128i a);
VCVTQQ2PS __m128 __mm_mask_cvtepi64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTQQ2PS __m128 __mm_maskz_cvtepi64_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.

## VCVTSD2SH—Convert Low FP64 Value to an FP16 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|---|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.F2.MAP5.W1 5A /r<br>VCVTSD2SH xmm1{k1}{z}, xmm2,<br>xmm3/m64 {er} | A         | V/V                          | AVX512-FP16           | Convert the low FP64 value in xmm3/m64 to an FP16 value and store the result in the low element of xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction converts the low FP64 value in the second source operand to an FP16 value, and stores the result in the low element of the destination operand.

When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

#### VCVTSD2SH dest, src1, src2

IF \*SRC2 is a register\* and (EVEX.b = 1):

```
SET_RM(EVEX.RC)
```

ELSE:

```
SET_RM(MXCSR.RC)
```

IF k1[0] OR \*no writemask\*:

```
DEST.fp16[0] := Convert_fp64_to_fp16(SRC2.fp64[0])
```

ELSE IF \*zeroing\*:

```
DEST.fp16[0] := 0
```

// else dest.fp16[0] remains unchanged

```
DEST[127:16] := SRC1[127:16]
```

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

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

```
VCVTSD2SH __m128h __mm_cvt_roundsd_sh (__m128h a, __m128d b, const int rounding);
```

```
VCVTSD2SH __m128h __mm_mask_cvt_roundsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b, const int rounding);
```

```
VCVTSD2SH __m128h __mm_maskz_cvt_roundsd_sh (__mmask8 k, __m128h a, __m128d b, const int rounding);
```

```
VCVTSD2SH __m128h __mm_cvtsd_sh (__m128h a, __m128d b);
```

```
VCVTSD2SH __m128h __mm_mask_cvtsd_sh (__m128h src, __mmask8 k, __m128h a, __m128d b);
```

```
VCVTSD2SH __m128h __mm_maskz_cvtsd_sh (__mmask8 k, __m128h a, __m128d b);
```

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

## VCVTSD2USI—Convert Scalar Double Precision Floating-Point Value to Unsigned Doubleword Integer

| Opcode/<br>Instruction                                   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.F2.OF.W0 79 /r<br>VCVTSD2USI r32, xmm1/m64{er} | A         | V/V                          | AVX512F                  | Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.                  |
| EVEX.LLIG.F2.OF.W1 79 /r<br>VCVTSD2USI r64, xmm1/m64{er} | A         | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64. |

### NOTES:

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

### Instruction Operand Encoding

| Op/En | Tuple Type   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Fixed | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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.”

## VCVTSH2SD—Convert Low FP16 Value to an FP64 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|--|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.F3.MAP5.W0 5A /r<br>VCVTSH2SD xmm1{k1}{z}, xmm2,<br>xmm3/m16 {sae} | A         | V/V                          | AVX512-FP16           | Convert the low FP16 value in xmm3/m16 to an FP64 value and store the result in the low element of xmm1 subject to writemask k1. Bits 127:64 of xmm2 are copied to xmm1[127:64]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction converts the low FP16 element in the second source operand to a FP64 element in the low element of the destination operand.

Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP64 element of the destination is updated according to the writemask.

### Operation

#### VCVTSH2SD dest, src1, src2

IF k1[0] OR \*no writemask\*:

```
DEST.fp64[0] := Convert_fp16_to_fp64(SRC2.fp16[0])
```

ELSE IF \*zeroing\*:

```
DEST.fp64[0] := 0
```

// else dest.fp64[0] remains unchanged

```
DEST[127:64] := SRC1[127:64]
```

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

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

```
VCVTSH2SD __m128d __mm_cvt_roundsh_sd (__m128d a, __m128h b, const int sae);
```

```
VCVTSH2SD __m128d __mm_mask_cvt_roundsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b, const int sae);
```

```
VCVTSH2SD __m128d __mm_maskz_cvt_roundsh_sd (__mmask8 k, __m128d a, __m128h b, const int sae);
```

```
VCVTSH2SD __m128d __mm_cvtsh_sd (__m128d a, __m128h b);
```

```
VCVTSH2SD __m128d __mm_mask_cvtsh_sd (__m128d src, __mmask8 k, __m128d a, __m128h b);
```

```
VCVTSH2SD __m128d __mm_maskz_cvtsh_sd (__mmask8 k, __m128d a, __m128h b);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal.

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

## VCVTSH2SI—Convert Low FP16 Value to Signed Integer

| Opcode/<br>Instruction                                     | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F3.MAP5.W0 2D /r<br>VCVTSH2SI r32, xmm1/m16 {er} | A         | V/V <sup>1</sup>             | AVX512-FP16           | Convert the low FP16 element in xmm1/m16 to a signed integer and store the result in r32. |
| EVEX.LLIG.F3.MAP5.W1 2D /r<br>VCVTSH2SI r64, xmm1/m16 {er} | A         | V/N.E.                       | AVX512-FP16           | Convert the low FP16 element in xmm1/m16 to a signed integer and store the result in r64. |

### NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------|---------------|---------------|-----------|-----------|
| A     | Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose 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 indefinite value is returned.

### Operation

#### VCVTSH2SI dest, src

IF \*SRC is a register\* and (EVEX.b = 1):

```
SET_RM(EVEX.RC)
```

ELSE:

```
SET_RM(MXCSR.RC)
```

IF 64-mode and OperandSize == 64:

```
DEST.qword := Convert_fp16_to_integer64(SRC.fp16[0])
```

ELSE:

```
DEST.dword := Convert_fp16_to_integer32(SRC.fp16[0])
```

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

```
VCVTSH2SI int __mm_cvt_roundsh_i32 (__m128h a, int rounding);
```

```
VCVTSH2SI __int64 __mm_cvt_roundsh_i64 (__m128h a, int rounding);
```

```
VCVTSH2SI int __mm_cvtsh_i32 (__m128h a);
```

```
VCVTSH2SI __int64 __mm_cvtsh_i64 (__m128h a);
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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



## VCVTSH2SS—Convert Low FP16 Value to FP32 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.NP.MAP6.W0 13 /r<br>VCVTSH2SS xmm1{k1}{z}, xmm2,<br>xmm3/m16 {sae} | A         | V/V                          | AVX512-FP16           | Convert the low FP16 element in xmm3/m16 to an FP32 value and store in the low element of xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction converts the low FP16 element in the second source operand to the low FP32 element of the destination operand.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

#### VCVTSH2SS dest, src1, src2

IF k1[0] OR \*no writemask\*:

```
DEST.fp32[0] := Convert_fp16_to_fp32(SRC2.fp16[0])
```

ELSE IF \*zeroing\*:

```
DEST.fp32[0] := 0
```

// else dest.fp32[0] remains unchanged

```
DEST[127:32] := SRC1[127:32]
```

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

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

```
VCVTSH2SS __m128 __mm_cvt_roundsh_ss (__m128 a, __m128h b, const int sae);
```

```
VCVTSH2SS __m128 __mm_mask_cvt_roundsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b, const int sae);
```

```
VCVTSH2SS __m128 __mm_maskz_cvt_roundsh_ss (__mmask8 k, __m128 a, __m128h b, const int sae);
```

```
VCVTSH2SS __m128 __mm_cvtsh_ss (__m128 a, __m128h b);
```

```
VCVTSH2SS __m128 __mm_mask_cvtsh_ss (__m128 src, __mmask8 k, __m128 a, __m128h b);
```

```
VCVTSH2SS __m128 __mm_maskz_cvtsh_ss (__mmask8 k, __m128 a, __m128h b);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal.

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

## VCVTSH2USI—Convert Low FP16 Value to Unsigned Integer

| Opcode/<br>Instruction                                      | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|---|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.F3.MAP5.W0 79 /r<br>VCVTSH2USI r32, xmm1/m16 {er} | A         | V/V <sup>1</sup>             | AVX512-FP16           | Convert the low FP16 element in xmm1/m16 to an unsigned integer and store the result in r32. |
| EVEX.LLIG.F3.MAP5.W1 79 /r<br>VCVTSH2USI r64, xmm1/m16 {er} | A         | V/N.E.                       | AVX512-FP16           | Convert the low FP16 element in xmm1/m16 to an unsigned integer and store the result in r64. |

### NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------|---------------|---------------|-----------|-----------|
| A     | Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose 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 indefinite value is returned.

### Operation

#### VCVTSH2USI dest, src

// SET\_RM() sets the rounding mode used for this instruction.

IF \*SRC is a register\* and (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:

DEST.qword := Convert\_fp16\_to\_unsigned\_integer64(SRC.fp16[0])

ELSE:

DEST.dword := Convert\_fp16\_to\_unsigned\_integer32(SRC.fp16[0])

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

VCVTSH2USI unsigned int \_\_mm\_cvt\_roundsh\_u32 (\_\_m128h a, int sae);

VCVTSH2USI unsigned \_\_int64 \_\_mm\_cvt\_roundsh\_u64 (\_\_m128h a, int rounding);

VCVTSH2USI unsigned int \_\_mm\_cvtsh\_u32 (\_\_m128h a);

VCVTSH2USI unsigned \_\_int64 \_\_mm\_cvtsh\_u64 (\_\_m128h a);

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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

## VCVTSI2SH—Convert a Signed Doubleword/Quadword Integer to an FP16 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|---|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F3.MAP5.W0 2A /r<br>VCVTSI2SH xmm1, xmm2, r32/m32<br>{er} | A         | V/V <sup>1</sup>             | AVX512-FP16           | Convert the signed doubleword integer in r32/<br>m32 to an FP16 value and store the result in<br>xmm1. Bits 127:16 of xmm2 are copied to<br>xmm1[127:16]. |
| EVEX.LLIG.F3.MAP5.W1 2A /r<br>VCVTSI2SH xmm1, xmm2, r64/m64<br>{er} | A         | V/N.E.                       | AVX512-FP16           | Convert the signed quadword integer in r64/m64<br>to an FP16 value and store the result in xmm1.<br>Bits 127:16 of xmm2 are copied to<br>xmm1[127:16].    |

### NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the second source operand to an FP16 value in the destination operand. The result is stored in the low word of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

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 127:16 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.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

### Operation

#### VCVTSI2SH dest, src1, src2

IF \*SRC2 is a register\* and (EVEX.b = 1):

```
SET_RM(EVEX.RC)
```

ELSE:

```
SET_RM(MXCSR.RC)
```

IF 64-mode and OperandSize == 64:

```
DEST.fp16[0] := Convert_integer64_to_fp16(SRC2.qword)
```

ELSE:

```
DEST.fp16[0] := Convert_integer32_to_fp16(SRC2.dword)
```

```
DEST[127:16] := SRC1[127:16]
```

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

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

```
VCVTSI2SH __m128h __mm_cvt_roundi32_sh (__m128h a, int b, int rounding);
```

```
VCVTSI2SH __m128h __mm_cvt_roundi64_sh (__m128h a, __int64 b, int rounding);
```

```
VCVTSI2SH __m128h __mm_cvti32_sh (__m128h a, int b);
```

```
VCVTSI2SH __m128h __mm_cvti64_sh (__m128h a, __int64 b);
```

**SIMD Floating-Point Exceptions**

Overflow, Precision.

**Other Exceptions**

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

## VCVTSS2SH—Convert Low FP32 Value to an FP16 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|---|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.NP.MAP5.WO 1D /r<br>VCVTSS2SH xmm1{k1}{z}, xmm2,<br>xmm3/m32 {er} | A         | V/V                          | AVX512-FP16           | Convert low FP32 value in xmm3/m32 to an FP16 value and store in the low element of xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction converts the low FP32 value in the second source operand to a FP16 value in the low element of the destination operand.

When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

#### VCVTSS2SH dest, src1, src2

IF \*SRC2 is a register\* and (EVEX.b = 1):

```
SET_RM(EVEX.RC)
```

ELSE:

```
SET_RM(MXCSR.RC)
```

IF k1[0] OR \*no writemask\*:

```
DEST.fp16[0] := Convert_fp32_to_fp16(SRC2.fp32[0])
```

ELSE IF \*zeroing\*:

```
DEST.fp16[0] := 0
```

// else dest.fp16[0] remains unchanged

```
DEST[127:16] := SRC1[127:16]
```

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

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

```
VCVTSS2SH __m128h __mm_cvt_roundss_sh (__m128h a, __m128 b, const int rounding);
```

```
VCVTSS2SH __m128h __mm_mask_cvt_roundss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b, const int rounding);
```

```
VCVTSS2SH __m128h __mm_maskz_cvt_roundss_sh (__mmask8 k, __m128h a, __m128 b, const int rounding);
```

```
VCVTSS2SH __m128h __mm_cvtss_sh (__m128h a, __m128 b);
```

```
VCVTSS2SH __m128h __mm_mask_cvtss_sh (__m128h src, __mmask8 k, __m128h a, __m128 b);
```

```
VCVTSS2SH __m128h __mm_maskz_cvtss_sh (__mmask8 k, __m128h a, __m128 b);
```

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

### Other Exceptions

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

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

| Opcode/<br>Instruction                                   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.F3.OF.W0 79 /r<br>VCVTSS2USI r32, xmm1/m32{er} | A         | V/V                          | AVX512F                  | Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32. |
| EVEX.LLIG.F3.OF.W1 79 /r<br>VCVTSS2USI r64, xmm1/m32{er} | A         | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.   |

### NOTES:

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

### Instruction Operand Encoding

| Op/En | Tuple Type   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Fixed | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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.”

## VCVTTPD2QQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Quadword Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F.W1 7A /r<br>VCVTTPD2QQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst      | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed double precision floating-point values from zmm2/m128/m64bcst to two packed quadword integers in zmm1 using truncation with writemask k1.   |
| EVEX.256.66.0F.W1 7A /r<br>VCVTTPD2QQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst      | A         | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed double precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 using truncation with writemask k1. |
| EVEX.512.66.0F.W1 7A /r<br>VCVTTPD2QQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae} | A         | V/V                          | AVX512DQ                 | Convert eight packed double precision floating-point values from zmm2/m512 to eight packed quadword integers in zmm1 using truncation with writemask k1.       |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

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

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

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

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

### Operation

#### VCVTTPD2QQ (EVEX Encoded Version) When SRC Operand is a Register

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

FOR j := 0 TO KL-1

  i := j \* 64

  IF k1[j] OR \*no writemask\*

    THEN DEST[+63:i] :=

      Convert\_Double\_Precision\_Floating\_Point\_To\_QuadInteger\_Truncate(SRC[+63:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[+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)

```

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

```

VCVTTPD2QQ __m512i __mm512_cvttpd_epi64( __m512d a);
VCVTTPD2QQ __m512i __mm512_mask_cvttpd_epi64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i __mm512_maskz_cvttpd_epi64( __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i __mm512_cvtt_roundpd_epi64( __m512d a, int sae);
VCVTTPD2QQ __m512i __mm512_mask_cvtt_roundpd_epi64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m512i __mm512_maskz_cvtt_roundpd_epi64( __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m256i __mm256_mask_cvttpd_epi64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i __mm256_maskz_cvttpd_epi64( __mmask8 k, __m256d a);
VCVTTPD2QQ __m128i __mm_mask_cvttpd_epi64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i __mm_maskz_cvttpd_epi64( __mmask8 k, __m128d a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

#UD If EVEX.vvvv != 1111B.

## VCVTTPD2UDQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Doubleword Integers

| Opcode Instruction  | Op/En | 64/32 Bit Mode Support | CPUID Feature Flag  | Description  |
|---|-------|------------------------|---------------------|--|
| EVEX.128.0F.W1 78 /r<br>VCVTTPD2UDQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst      | A     | V/V                    | AVX512VL<br>AVX512F | 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.     |
| EVEX.256.0F.W1 78 02 /r<br>VCVTTPD2UDQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst   | A     | V/V                    | AVX512VL<br>AVX512F | 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.   |
| EVEX.512.0F.W1 78 /r<br>VCVTTPD2UDQ ymm1 {k1}{z},<br>zmm2/m512/m64bcst{sae} | A     | V/V                    | AVX512F             | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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

      DEST[i+31:i] :=

      Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[k+63:k])

    ELSE

      IF \*merging-masking\* ; merging-masking

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

        ELSE ; zeroing-masking

          DEST[i+31:i] := 0

    FI

FI;

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

### VCVTTPD2UDQ (EVEX Encoded Versions) When SRC Operand is a Memory Source

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

```
FOR j := 0 TO KL-1
  i := j * 32
  k := j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+31:i] :=
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        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_cvttpd_epu32( __m512d a);
VCVTTPD2UDQ __m256i _mm512_mask_cvttpd_epu32( __m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_maskz_cvttpd_epu32( __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_cvtt_roundpd_epu32( __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_mask_cvtt_roundpd_epu32( __m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_maskz_cvtt_roundpd_epu32( __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m128i _mm256_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm256_maskz_cvttpd_epu32( __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm_mask_cvttpd_epu32( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __m128i _mm_maskz_cvttpd_epu32( __mmask8 k, __m128d a);
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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

Additionally:

#UD                    If EVEX.vvvv != 1111B.

## VCVTPD2UQQ—Convert With Truncation Packed Double Precision Floating-Point Values to Packed Unsigned Quadword Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F.W1 78 /r<br>VCVTPD2UQQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst      | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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.   |
| EVEX.256.66.0F.W1 78 /r<br>VCVTPD2UQQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst      | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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. |
| EVEX.512.66.0F.W1 78 /r<br>VCVTPD2UQQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae} | A         | V/V                          | AVX512DQ                 | Convert eight packed double precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.        |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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

#### VCVTPD2UQQ (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[j] 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_cvttpd_epu64( __m512d a);
VCVTTPD2UQQ __m512i __mm512_mask_cvttpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_maskz_cvttpd_epu64( __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i __mm512_cvtt_roundpd_epu64( __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_mask_cvtt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i __mm512_maskz_cvtt_roundpd_epu64( __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i __mm256_mask_cvttpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i __mm256_maskz_cvttpd_epu64( __mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i __mm_mask_cvttpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i __mm_maskz_cvttpd_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.

## VCVTTPH2DQ—Convert with Truncation Packed FP16 Values to Signed Doubleword Integers

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.F3.MAP5.W0 5B /r<br>VCVTTPH2DQ xmm1{k1}{z}, xmm2/<br>m64/m16bcst        | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed FP16 values in xmm2/m64/<br>m16bcst to four signed doubleword integers, and<br>store the result in xmm1 using truncation subject<br>to writemask k1.        |
| EVEX.256.F3.MAP5.W0 5B /r<br>VCVTTPH2DQ ymm1{k1}{z}, xmm2/<br>m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight signed doubleword<br>integers, and store the result in ymm1 using<br>truncation subject to writemask k1.     |
| EVEX.512.F3.MAP5.W0 5B /r<br>VCVTTPH2DQ zmm1{k1}{z}, ymm2/<br>m256/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Convert sixteen packed FP16 values in ymm2/<br>m256/m16bcst to sixteen signed doubleword<br>integers, and store the result in zmm1 using<br>truncation subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Half  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to signed doubleword integers in destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTTPH2DQ dest, src**

VL = 128, 256 or 512

KL := VL / 32

FOR j := 0 TO KL-1:

  IF k1[j] OR \*no writemask\*:

    IF \*SRC is memory\* and EVEX.b = 1:

      tsrc := SRC.fp16[0]

    ELSE

      tsrc := SRC.fp16[j]

      DEST.fp32[j] := Convert\_fp16\_to\_integer32\_truncate(tsrc)

    ELSE IF \*zeroing\*:

      DEST.fp32[j] := 0

    // else dest.fp32[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTTPH2DQ \_\_m512i \_mm512\_cvtt\_roundph\_epi32 (\_\_m256h a, int sae);  
 VCVTTPH2DQ \_\_m512i \_mm512\_mask\_cvtt\_roundph\_epi32 (\_\_m512i src, \_\_mmask16 k, \_\_m256h a, int sae);  
 VCVTTPH2DQ \_\_m512i \_mm512\_maskz\_cvtt\_roundph\_epi32 (\_\_mmask16 k, \_\_m256h a, int sae);  
 VCVTTPH2DQ \_\_m128i \_mm\_cvttph\_epi32 (\_\_m128h a);  
 VCVTTPH2DQ \_\_m128i \_mm\_mask\_cvttph\_epi32 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2DQ \_\_m128i \_mm\_maskz\_cvttph\_epi32 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2DQ \_\_m256i \_mm256\_cvttph\_epi32 (\_\_m128h a);  
 VCVTTPH2DQ \_\_m256i \_mm256\_mask\_cvttph\_epi32 (\_\_m256i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2DQ \_\_m256i \_mm256\_maskz\_cvttph\_epi32 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2DQ \_\_m512i \_mm512\_cvttph\_epi32 (\_\_m256h a);  
 VCVTTPH2DQ \_\_m512i \_mm512\_mask\_cvttph\_epi32 (\_\_m512i src, \_\_mmask16 k, \_\_m256h a);  
 VCVTTPH2DQ \_\_m512i \_mm512\_maskz\_cvttph\_epi32 (\_\_mmask16 k, \_\_m256h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

## VCVTTPH2QQ—Convert with Truncation Packed FP16 Values to Signed Quadword Integers

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP5.W0 7A /r<br>VCVTTPH2QQ xmm1{k1}{z}, xmm2/<br>m32/m16bcst        | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert two packed FP16 values in xmm2/m32/<br>m16bcst to two signed quadword integers, and<br>store the result in xmm1 using truncation subject<br>to writemask k1.      |
| EVEX.256.66.MAP5.W0 7A /r<br>VCVTTPH2QQ ymm1{k1}{z}, xmm2/<br>m64/m16bcst        | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed FP16 values in xmm2/m64/<br>m16bcst to four signed quadword integers, and<br>store the result in ymm1 using truncation subject<br>to writemask k1.    |
| EVEX.512.66.MAP5.W0 7A /r<br>VCVTTPH2QQ zmm1{k1}{z}, xmm2/<br>m128/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight signed quadword<br>integers, and store the result in zmm1 using<br>truncation subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------|---------------|---------------|-----------|-----------|
| A     | Quarter | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to 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 is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTTPH2QQ dest, src**

VL = 128, 256 or 512

KL := VL / 64

FOR j := 0 TO KL-1:

  IF k1[j] OR \*no writemask\*:

    IF \*SRC is memory\* and EVEX.b = 1:

      tsrc := SRC.fp16[0]

    ELSE

      tsrc := SRC.fp16[j]

      DEST.qword[j] := Convert\_fp16\_to\_integer64\_truncate(tsrc)

    ELSE IF \*zeroing\*:

      DEST.qword[j] := 0

    // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0



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

VCVTTPH2QQ \_\_m512i \_\_mm512\_cvtt\_roundph\_epi64 (\_\_m128h a, int sae);  
 VCVTTPH2QQ \_\_m512i \_\_mm512\_mask\_cvtt\_roundph\_epi64 (\_\_m512i src, \_\_mmask8 k, \_\_m128h a, int sae);  
 VCVTTPH2QQ \_\_m512i \_\_mm512\_maskz\_cvtt\_roundph\_epi64 (\_\_mmask8 k, \_\_m128h a, int sae);  
 VCVTTPH2QQ \_\_m128i \_\_mm\_cvttph\_epi64 (\_\_m128h a);  
 VCVTTPH2QQ \_\_m128i \_\_mm\_mask\_cvttph\_epi64 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2QQ \_\_m128i \_\_mm\_maskz\_cvttph\_epi64 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2QQ \_\_m256i \_\_mm256\_cvttph\_epi64 (\_\_m128h a);  
 VCVTTPH2QQ \_\_m256i \_\_mm256\_mask\_cvttph\_epi64 (\_\_m256i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2QQ \_\_m256i \_\_mm256\_maskz\_cvttph\_epi64 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2QQ \_\_m512i \_\_mm512\_cvttph\_epi64 (\_\_m128h a);  
 VCVTTPH2QQ \_\_m512i \_\_mm512\_mask\_cvttph\_epi64 (\_\_m512i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2QQ \_\_m512i \_\_mm512\_maskz\_cvttph\_epi64 (\_\_mmask8 k, \_\_m128h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

## VCVTTPH2UDQ—Convert with Truncation Packed FP16 Values to Unsigned Doubleword Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.MAP5.W0 78 /r<br>VCVTTPH2UDQ xmm1{k1}{z}, xmm2/<br>m64/m16bcst        | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed FP16 values in xmm2/m64/<br>m16bcst to four unsigned doubleword integers,<br>and store the result in xmm1 using truncation<br>subject to writemask k1.        |
| EVEX.256.NP.MAP5.W0 78 /r<br>VCVTTPH2UDQ ymm1{k1}{z}, xmm2/<br>m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight unsigned doubleword<br>integers, and store the result in ymm1 using<br>truncation subject to writemask k1.     |
| EVEX.512.NP.MAP5.W0 78 /r<br>VCVTTPH2UDQ zmm1{k1}{z}, ymm2/<br>m256/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Convert sixteen packed FP16 values in ymm2/<br>m256/m16bcst to sixteen unsigned doubleword<br>integers, and store the result in zmm1 using<br>truncation subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Half  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to 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 indefinite value is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTTPH2UDQ dest, src**

VL = 128, 256 or 512

KL := VL / 32

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.fp16[0]

ELSE

tsrc := SRC.fp16[j]

DEST.dword[j] := Convert\_fp16\_to\_unsigned\_integer32\_truncate(tsrc)

ELSE IF \*zeroing\*:

DEST.dword[j] := 0

// else dest.dword[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTTPH2UDQ \_\_m512i \_mm512\_cvtt\_roundph\_epu32 (\_\_m256h a, int sae);  
 VCVTTPH2UDQ \_\_m512i \_mm512\_mask\_cvtt\_roundph\_epu32 (\_\_m512i src, \_\_mmask16 k, \_\_m256h a, int sae);  
 VCVTTPH2UDQ \_\_m512i \_mm512\_maskz\_cvtt\_roundph\_epu32 (\_\_mmask16 k, \_\_m256h a, int sae);  
 VCVTTPH2UDQ \_\_m128i \_mm\_cvttph\_epu32 (\_\_m128h a);  
 VCVTTPH2UDQ \_\_m128i \_mm\_mask\_cvttph\_epu32 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UDQ \_\_m128i \_mm\_maskz\_cvttph\_epu32 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UDQ \_\_m256i \_mm256\_cvttph\_epu32 (\_\_m128h a);  
 VCVTTPH2UDQ \_\_m256i \_mm256\_mask\_cvttph\_epu32 (\_\_m256i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UDQ \_\_m256i \_mm256\_maskz\_cvttph\_epu32 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UDQ \_\_m512i \_mm512\_cvttph\_epu32 (\_\_m256h a);  
 VCVTTPH2UDQ \_\_m512i \_mm512\_mask\_cvttph\_epu32 (\_\_m512i src, \_\_mmask16 k, \_\_m256h a);  
 VCVTTPH2UDQ \_\_m512i \_mm512\_maskz\_cvttph\_epu32 (\_\_mmask16 k, \_\_m256h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

**VCVTTPH2UQQ—Convert with Truncation Packed FP16 Values to Unsigned Quadword Integers**

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP5.W0 78 /r<br>VCVTTPH2UQQ xmm1{k1}{z},<br>xmm2/m32/m16bcst         | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert two packed FP16 values in xmm2/m32/<br>m16bcst to two unsigned quadword integers,<br>and store the result in xmm1 using truncation<br>subject to writemask k1.      |
| EVEX.256.66.MAP5.W0 78 /r<br>VCVTTPH2UQQ ymm1{k1}{z},<br>xmm2/m64/m16bcst         | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed FP16 values in xmm2/m64/<br>m16bcst to four unsigned quadword integers,<br>and store the result in ymm1 using truncation<br>subject to writemask k1.    |
| EVEX.512.66.MAP5.W0 78 /r<br>VCVTTPH2UQQ zmm1{k1}{z}, xmm2/<br>m128/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight unsigned quadword<br>integers, and store the result in zmm1 using<br>truncation subject to writemask k1. |

**Instruction Operand Encoding**

| Op/En | Tuple   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------|---------------|---------------|-----------|-----------|
| A     | Quarter | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

**Description**

This instruction converts packed FP16 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 indefinite value is returned.

The destination elements are updated according to the writemask.

**Operation**

**VCVTTPH2UQQ dest, src**

VL = 128, 256 or 512

KL := VL / 64

FOR j := 0 TO KL-1:

  IF k1[j] OR \*no writemask\*:

    IF \*SRC is memory\* and EVEX.b = 1:

      tsrc := SRC.fp16[0]

    ELSE

      tsrc := SRC.fp16[j]

    DEST.qword[j] := Convert\_fp16\_to\_unsigned\_integer64\_truncate(tsrc)

  ELSE IF \*zeroing\*:

    DEST.qword[j] := 0

  // else dest.qword[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTTPH2UQQ \_\_m512i \_\_mm512\_cvtt\_roundph\_epu64 (\_\_m128h a, int sae);  
 VCVTTPH2UQQ \_\_m512i \_\_mm512\_mask\_cvtt\_roundph\_epu64 (\_\_m512i src, \_\_mmask8 k, \_\_m128h a, int sae);  
 VCVTTPH2UQQ \_\_m512i \_\_mm512\_maskz\_cvtt\_roundph\_epu64 (\_\_mmask8 k, \_\_m128h a, int sae);  
 VCVTTPH2UQQ \_\_m128i \_\_mm\_cvttph\_epu64 (\_\_m128h a);  
 VCVTTPH2UQQ \_\_m128i \_\_mm\_mask\_cvttph\_epu64 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UQQ \_\_m128i \_\_mm\_maskz\_cvttph\_epu64 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UQQ \_\_m256i \_\_mm256\_cvttph\_epu64 (\_\_m128h a);  
 VCVTTPH2UQQ \_\_m256i \_\_mm256\_mask\_cvttph\_epu64 (\_\_m256i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UQQ \_\_m256i \_\_mm256\_maskz\_cvttph\_epu64 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UQQ \_\_m512i \_\_mm512\_cvttph\_epu64 (\_\_m128h a);  
 VCVTTPH2UQQ \_\_m512i \_\_mm512\_mask\_cvttph\_epu64 (\_\_m512i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UQQ \_\_m512i \_\_mm512\_maskz\_cvttph\_epu64 (\_\_mmask8 k, \_\_m128h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

## VCVTTPH2UW—Convert Packed FP16 Values to Unsigned Word Integers

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.MAP5.W0 7C /r<br>VCVTTPH2UW xmm1{k1}{z}, xmm2/<br>m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight unsigned word integers,<br>and store the result in xmm1 using truncation<br>subject to writemask k1.           |
| EVEX.256.NP.MAP5.W0 7C /r<br>VCVTTPH2UW ymm1{k1}{z}, ymm2/<br>m256/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert sixteen packed FP16 values in ymm2/<br>m256/m16bcst to sixteen unsigned word<br>integers, and store the result in ymm1 using<br>truncation subject to writemask k1.       |
| EVEX.512.NP.MAP5.W0 7C /r<br>VCVTTPH2UW zmm1{k1}{z}, zmm2/<br>m512/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Convert thirty-two packed FP16 values in zmm2/<br>m512/m16bcst to thirty-two unsigned word<br>integers, and store the result in zmm1 using<br>truncation subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to unsigned word 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 indefinite value is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTTPH2UW** dest, src

VL = 128, 256 or 512

KL := VL / 16

FOR j := 0 TO KL-1:

  IF k1[j] OR \*no writemask\*:

    IF \*SRC is memory\* and EVEX.b = 1:

      tsrc := SRC.fp16[0]

    ELSE

      tsrc := SRC.fp16[j]

      DEST.word[j] := Convert\_fp16\_to\_unsigned\_integer16\_truncate(tsrc)

    ELSE IF \*zeroing\*:

      DEST.word[j] := 0

    // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTTPH2UW \_\_m512i \_\_mm512\_cvtt\_roundph\_epu16 (\_\_m512h a, int sae);  
 VCVTTPH2UW \_\_m512i \_\_mm512\_mask\_cvtt\_roundph\_epu16 (\_\_m512i src, \_\_mmask32 k, \_\_m512h a, int sae);  
 VCVTTPH2UW \_\_m512i \_\_mm512\_maskz\_cvtt\_roundph\_epu16 (\_\_mmask32 k, \_\_m512h a, int sae);  
 VCVTTPH2UW \_\_m128i \_\_mm\_cvttph\_epu16 (\_\_m128h a);  
 VCVTTPH2UW \_\_m128i \_\_mm\_mask\_cvttph\_epu16 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UW \_\_m128i \_\_mm\_maskz\_cvttph\_epu16 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2UW \_\_m256i \_\_mm256\_cvttph\_epu16 (\_\_m256h a);  
 VCVTTPH2UW \_\_m256i \_\_mm256\_mask\_cvttph\_epu16 (\_\_m256i src, \_\_mmask16 k, \_\_m256h a);  
 VCVTTPH2UW \_\_m256i \_\_mm256\_maskz\_cvttph\_epu16 (\_\_mmask16 k, \_\_m256h a);  
 VCVTTPH2UW \_\_m512i \_\_mm512\_cvttph\_epu16 (\_\_m512h a);  
 VCVTTPH2UW \_\_m512i \_\_mm512\_mask\_cvttph\_epu16 (\_\_m512i src, \_\_mmask32 k, \_\_m512h a);  
 VCVTTPH2UW \_\_m512i \_\_mm512\_maskz\_cvttph\_epu16 (\_\_mmask32 k, \_\_m512h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

## VCVTTPH2W—Convert Packed FP16 Values to Signed Word Integers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP5.W0 7C /r<br>VCVTTPH2W xmm1{k1}{z}, xmm2/<br>m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed FP16 values in xmm2/<br>m128/m16bcst to eight signed word integers,<br>and store the result in xmm1 using truncation<br>subject to writemask k1.           |
| EVEX.256.66.MAP5.W0 7C /r<br>VCVTTPH2W ymm1{k1}{z}, ymm2/<br>m256/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert sixteen packed FP16 values in ymm2/<br>m256/m16bcst to sixteen signed word integers,<br>and store the result in ymm1 using truncation<br>subject to writemask k1.       |
| EVEX.512.66.MAP5.W0 7C /r<br>VCVTTPH2W zmm1{k1}{z}, zmm2/<br>m512/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Convert thirty-two packed FP16 values in zmm2/<br>m512/m16bcst to thirty-two signed word<br>integers, and store the result in zmm1 using<br>truncation subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed FP16 values in the source operand to signed word 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 indefinite value is returned.

The destination elements are updated according to the writemask.

### Operation

**VCVTTPH2W dest, src**

VL = 128, 256 or 512

KL := VL / 16

FOR j := 0 TO KL-1:

  IF k1[j] OR \*no writemask\*:

    IF \*SRC is memory\* and EVEX.b = 1:

      tsrc := SRC.fp16[0]

    ELSE

      tsrc := SRC.fp16[j]

    DEST.word[j] := Convert\_fp16\_to\_integer16\_truncate(tsrc)

  ELSE IF \*zeroing\*:

    DEST.word[j] := 0

  // else dest.word[j] remains unchanged

DEST[MAXVL-1:VL] := 0



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

VCVTTPH2W \_\_m512i \_\_mm512\_cvtt\_roundph\_epi16 (\_\_m512h a, int sae);  
 VCVTTPH2W \_\_m512i \_\_mm512\_mask\_cvtt\_roundph\_epi16 (\_\_m512i src, \_\_mmask32 k, \_\_m512h a, int sae);  
 VCVTTPH2W \_\_m512i \_\_mm512\_maskz\_cvtt\_roundph\_epi16 (\_\_mmask32 k, \_\_m512h a, int sae);  
 VCVTTPH2W \_\_m128i \_\_mm\_cvttph\_epi16 (\_\_m128h a);  
 VCVTTPH2W \_\_m128i \_\_mm\_mask\_cvttph\_epi16 (\_\_m128i src, \_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2W \_\_m128i \_\_mm\_maskz\_cvttph\_epi16 (\_\_mmask8 k, \_\_m128h a);  
 VCVTTPH2W \_\_m256i \_\_mm256\_cvttph\_epi16 (\_\_m256h a);  
 VCVTTPH2W \_\_m256i \_\_mm256\_mask\_cvttph\_epi16 (\_\_m256i src, \_\_mmask16 k, \_\_m256h a);  
 VCVTTPH2W \_\_m256i \_\_mm256\_maskz\_cvttph\_epi16 (\_\_mmask16 k, \_\_m256h a);  
 VCVTTPH2W \_\_m512i \_\_mm512\_cvttph\_epi16 (\_\_m512h a);  
 VCVTTPH2W \_\_m512i \_\_mm512\_mask\_cvttph\_epi16 (\_\_m512i src, \_\_mmask32 k, \_\_m512h a);  
 VCVTTPH2W \_\_m512i \_\_mm512\_maskz\_cvttph\_epi16 (\_\_mmask32 k, \_\_m512h a);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

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

## VCVTTPS2QQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Signed Quadword Integer Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F.W0 7A /r<br>VCVTTPS2QQ xmm1 {k1}{z},<br>xmm2/m64/m32bcst       | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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.      |
| EVEX.256.66.0F.W0 7A /r<br>VCVTTPS2QQ ymm1 {k1}{z},<br>xmm2/m128/m32bcst      | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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.   |
| EVEX.512.66.0F.W0 7A /r<br>VCVTTPS2QQ zmm1 {k1}{z},<br>ymm2/m256/m32bcst{sae} | A         | V/V                          | AVX512DQ                 | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Half       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

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

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

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

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

### Operation

#### VCVTTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Register

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

FOR j := 0 TO KL-1

  i := j \* 64

  k := j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[+63:i] :=

      Convert\_Single\_Precision\_To\_QuadInteger\_Truncate(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[+63:i] := 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VCVTTPS2QQ (EVEX Encoded Versions) When SRC Operand is a Memory Source**

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

```

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

```

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

```

VCVTTPS2QQ __m512i __mm512_cvttps_epi64( __m256 a);
VCVTTPS2QQ __m512i __mm512_mask_cvttps_epi64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i __mm512_maskz_cvttps_epi64( __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i __mm512_cvtt_roundps_epi64( __m256 a, int sae);
VCVTTPS2QQ __m512i __mm512_mask_cvtt_roundps_epi64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m512i __mm512_maskz_cvtt_roundps_epi64( __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i __mm256_mask_cvttps_epi64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m256i __mm256_maskz_cvttps_epi64( __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i __mm_mask_cvttps_epi64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __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.

## VCVTTPS2UDQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.OF.W0 78 /r<br>VCVTTPS2UDQ xmm1 {k1}{z},<br>xmm2/m128/m32bcst      | A         | V/V                          | AVX512VL<br>AVX512F      | 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.       |
| EVEX.256.OF.W0 78 /r<br>VCVTTPS2UDQ ymm1 {k1}{z},<br>ymm2/m256/m32bcst      | A         | V/V                          | AVX512VL<br>AVX512F      | 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.     |
| EVEX.512.OF.W0 78 /r<br>VCVTTPS2UDQ zmm1 {k1}{z},<br>zmm2/m512/m32bcst{sae} | A         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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 \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] :=

      Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[i+31:i])

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+31:i] := 0

  FI

FI;

ENDFOR

DEST[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.

## VCVTTPS2UQQ—Convert With Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F.W0 78 /r<br>VCVTTPS2UQQ xmm1 {k1}{z},<br>xmm2/m64/m32bcst       | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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.      |
| EVEX.256.66.0F.W0 78 /r<br>VCVTTPS2UQQ ymm1 {k1}{z},<br>xmm2/m128/m32bcst      | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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.   |
| EVEX.512.66.0F.W0 78 /r<br>VCVTTPS2UQQ zmm1 {k1}{z},<br>ymm2/m256/m32bcst{sae} | A         | V/V                          | AVX512DQ                 | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Half       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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

**VCVTTPS2UQQ (EVEX Encoded Versions) When SRC Operand is a Memory Source**

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

```

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

```

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

```

VCVTTPS2UQQ __mm<size>[_mask[z]]_cvttroundps_epu64
VCVTTPS2UQQ __m512i __mm512_cvttps_epu64( __m256 a);
VCVTTPS2UQQ __m512i __mm512_mask_cvttps_epu64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_maskz_cvttps_epu64( __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_cvttroundps_epu64( __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_mask_cvttroundps_epu64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_maskz_cvttroundps_epu64( __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m256i __mm256_mask_cvttps_epu64( __m256i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m256i __mm256_maskz_cvttps_epu64( __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i __mm_mask_cvttps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i __mm_maskz_cvttps_epu64( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

Additionally:

#UD If EVEX.vvvv != 1111B.

## VCVTTSD2USI—Convert With Truncation Scalar Double Precision Floating-Point Value to Unsigned Integer

| Opcode/<br>Instruction                                     | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.LLIG.F2.OF.W0 78 /r<br>VCVTTSD2USI r32, xmm1/m64{sae} | A         | V/V                          | AVX512F                  | Convert one double precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.                  |
| EVEX.LLIG.F2.OF.W1 78 /r<br>VCVTTSD2USI r64, xmm1/m64{sae} | A         | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one double precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation. |

### NOTES:

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

### Instruction Operand Encoding

| Op/En | Tuple Type   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Fixed | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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_cvttssd_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundssd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvttssd_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundssd_u64(__m128d, int sae);
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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



## VCVTTSH2SI—Convert with Truncation Low FP16 Value to a Signed Integer

| Opcode/<br>Instruction                                       | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F3.MAP5.W0 2C /r<br>VCVTTSH2SI r32, xmm1/m16 {sae} | A         | V/V <sup>1</sup>             | AVX512-FP16           | Convert FP16 value in the low element of xmm1/m16 to a signed integer and store the result in r32 using truncation. |
| EVEX.LLIG.F3.MAP5.W1 2C /r<br>VCVTTSH2SI r64, xmm1/m16 {sae} | A         | V/N.E.                       | AVX512-FP16           | Convert FP16 value in the low element of xmm1/m16 to a signed integer and store the result in r64 using truncation. |

### NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------|---------------|---------------|-----------|-----------|
| A     | Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts the low FP16 element in the source operand to a signed integer in the destination general purpose 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 indefinite value is returned.

### Operation

#### VCVTTSH2SI dest, src

IF 64-mode and OperandSize == 64:

```
DEST.qword := Convert_fp16_to_integer64_truncate(SRC.fp16[0])
```

ELSE:

```
DEST.dword := Convert_fp16_to_integer32_truncate(SRC.fp16[0])
```

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

```
VCVTTSH2SI int __mm_cvtt_roundsh_i32 (__m128h a, int sae);
```

```
VCVTTSH2SI __int64 __mm_cvtt_roundsh_i64 (__m128h a, int sae);
```

```
VCVTTSH2SI int __mm_cvttsh_i32 (__m128h a);
```

```
VCVTTSH2SI __int64 __mm_cvttsh_i64 (__m128h a);
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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

**VCVTTSH2USI—Convert with Truncation Low FP16 Value to an Unsigned Integer**

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|---|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.F3.MAP5.W0 78 /r<br>VCVTTSH2USI r32, xmm1/m16 {sae} | A         | V/V <sup>1</sup>             | AVX512-FP16           | Convert FP16 value in the low element of xmm1/m16 to an unsigned integer and store the result in r32 using truncation. |
| EVEX.LLIG.F3.MAP5.W1 78 /r<br>VCVTTSH2USI r64, xmm1/m16 {sae} | A         | V/N.E.                       | AVX512-FP16           | Convert FP16 value in the low element of xmm1/m16 to an unsigned integer and store the result in r64 using truncation. |

**NOTES:**

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

**Instruction Operand Encoding**

| Op/En | Tuple  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------|---------------|---------------|-----------|-----------|
| A     | Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

**Description**

This instruction converts the low FP16 element in the source operand to an unsigned integer in the destination general purpose 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 indefinite value is returned.

**Operation****VCVTTSH2USI dest, src**

IF 64-mode and OperandSize == 64:

```
DEST.qword := Convert_fp16_to_unsigned_integer64_truncate(SRC.fp16[0])
```

ELSE:

```
DEST.dword := Convert_fp16_to_unsigned_integer32_truncate(SRC.fp16[0])
```

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

```
VCVTTSH2USI unsigned int __mm_cvtt_roundsh_u32 (__m128h a, int sae);
```

```
VCVTTSH2USI unsigned __int64 __mm_cvtt_roundsh_u64 (__m128h a, int sae);
```

```
VCVTTSH2USI unsigned int __mm_cvttsh_u32 (__m128h a);
```

```
VCVTTSH2USI unsigned __int64 __mm_cvttsh_u64 (__m128h a);
```

**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

| Opcode/<br>Instruction                                     | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.LLIG.F3.OF.W0 78 /r<br>VCVTTSS2USI r32, xmm1/m32{sae} | A         | V/V                          | AVX512F                  | Convert one single precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation. |
| EVEX.LLIG.F3.OF.W1 78 /r<br>VCVTTSS2USI r64, xmm1/m32{sae} | A         | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one single precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.   |

### NOTES:

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

### Instruction Operand Encoding

| Op/En | Tuple Type   | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Fixed | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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\_cvtss\_u32( \_\_m128 a);

VCVTTSS2USI unsigned int \_\_mm\_cvt\_roundss\_u32( \_\_m128 a, int sae);

VCVTTSS2USI unsigned \_\_int64 \_\_mm\_cvtss\_u64( \_\_m128 a);

VCVTTSS2USI unsigned \_\_int64 \_\_mm\_cvt\_roundss\_u64( \_\_m128 a, int sae);

### SIMD Floating-Point Exceptions

Invalid, Precision.

### Other Exceptions

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

## VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.F3.0F.W0 7A /r<br>VCVTUDQ2PD xmm1 {k1}{z},<br>xmm2/m64/m32bcst  | A         | V/V                          | AVX512VL<br>AVX512F      | Convert two packed unsigned doubleword integers from ymm2/m64/m32bcst to packed double precision floating-point values in zmm1 with writemask k1.          |
| EVEX.256.F3.0F.W0 7A /r<br>VCVTUDQ2PD ymm1 {k1}{z},<br>xmm2/m128/m32bcst | A         | V/V                          | AVX512VL<br>AVX512F      | Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double precision floating-point values in zmm1 with writemask k1.        |
| EVEX.512.F3.0F.W0 7A /r<br>VCVTUDQ2PD zmm1 {k1}{z},<br>ymm2/m256/m32bcst | A         | V/V                          | AVX512F                  | Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Half       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

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

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

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

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

### Operation

#### VCVTUDQ2PD (EVEX Encoded Versions) When SRC Operand is a Register

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

FOR j := 0 TO KL-1

  i := j \* 64

  k := j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] :=

      Convert\_UIInteger\_To\_Double\_Precision\_Floating\_Point(SRC[k+31:k])

  ELSE

    IF \*merging-masking\* ; merging-masking

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

    ELSE ; zeroing-masking

      DEST[i+63:i] := 0

  FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VCVTUDQ2PD (EVEX Encoded Versions) When SRC Operand is a Memory Source**

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

```

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

```

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

```

VCVTUDQ2PD __m512d __mm512_cvtepu32_pd( __m256i a);
VCVTUDQ2PD __m512d __mm512_mask_cvtepu32_pd( __m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d __mm512_maskz_cvtepu32_pd( __mmask8 k, __m256i a);
VCVTUDQ2PD __m256d __mm256_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m256d __mm256_mask_cvtepu32_pd( __m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d __mm256_maskz_cvtepu32_pd( __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d __mm_cvtepu32_pd( __m128i a);
VCVTUDQ2PD __m128d __mm_mask_cvtepu32_pd( __m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d __mm_maskz_cvtepu32_pd( __mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-51, “Type E5 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.

## VCVTUDQ2PH—Convert Packed Unsigned Doubleword Integers to Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.F2.MAP5.W0 7A /r<br>VCVTUDQ2PH xmm1{k1}{z}, xmm2/<br>m128/m32bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.    |
| EVEX.256.F2.MAP5.W0 7A /r<br>VCVTUDQ2PH xmm1{k1}{z}, ymm2/<br>m256/m32bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.   |
| EVEX.512.F2.MAP5.W0 7A /r<br>VCVTUDQ2PH ymm1{k1}{z}, zmm2/<br>m512/m32bcst {er} | A         | V/V                          | AVX512-FP16             | Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to packed FP16 values, and store the result in ymm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed unsigned doubleword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

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

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

### Operation

**VCVTUDQ2PH dest, src**

VL = 128, 256 or 512

KL := VL / 32

IF \*SRC is a register\* and (VL = 512) AND (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.dword[0]

ELSE

tsrc := SRC.dword[j]

DEST.fp16[j] := Convert\_unsigned\_integer32\_to\_fp16(tsrc)

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

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

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

VCVTUDQ2PH \_\_m256h \_\_mm512\_cvt\_roundedu32\_ph (\_\_m512i a, int rounding);  
 VCVTUDQ2PH \_\_m256h \_\_mm512\_mask\_cvt\_roundedu32\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m512i a, int rounding);  
 VCVTUDQ2PH \_\_m256h \_\_mm512\_maskz\_cvt\_roundedu32\_ph (\_\_mmask16 k, \_\_m512i a, int rounding);  
 VCVTUDQ2PH \_\_m128h \_\_mm\_cvtepu32\_ph (\_\_m128i a);  
 VCVTUDQ2PH \_\_m128h \_\_mm\_mask\_cvtepu32\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128i a);  
 VCVTUDQ2PH \_\_m128h \_\_mm\_maskz\_cvtepu32\_ph (\_\_mmask8 k, \_\_m128i a);  
 VCVTUDQ2PH \_\_m128h \_\_mm256\_cvtepu32\_ph (\_\_m256i a);  
 VCVTUDQ2PH \_\_m128h \_\_mm256\_mask\_cvtepu32\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m256i a);  
 VCVTUDQ2PH \_\_m128h \_\_mm256\_maskz\_cvtepu32\_ph (\_\_mmask8 k, \_\_m256i a);  
 VCVTUDQ2PH \_\_m256h \_\_mm512\_cvtepu32\_ph (\_\_m512i a);  
 VCVTUDQ2PH \_\_m256h \_\_mm512\_mask\_cvtepu32\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m512i a);  
 VCVTUDQ2PH \_\_m256h \_\_mm512\_maskz\_cvtepu32\_ph (\_\_mmask16 k, \_\_m512i a);

**SIMD Floating-Point Exceptions**

Overflow, Precision.

**Other Exceptions**

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

## VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| EVEX.128.F2.0F.W0 7A /r<br>VCVTUDQ2PS xmm1 {k1}{z},<br>xmm2/m128/m32bcst     | A          | V/V                          | AVX512VL<br>AVX512F      | Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single precision floating-point values in xmm1 with writemask k1.            |
| EVEX.256.F2.0F.W0 7A /r<br>VCVTUDQ2PS ymm1 {k1}{z},<br>ymm2/m256/m32bcst     | A          | V/V                          | AVX512VL<br>AVX512F      | Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single precision floating-point values in zmm1 with writemask k1.           |
| EVEX.512.F2.0F.W0 7A /r<br>VCVTUDQ2PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{er} | A          | V/V                          | AVX512F                  | Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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_ULInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE
                ; zeroing-masking
                DEST[i+31:i] := 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```



**VCVTUDQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source**

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

```

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

```

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

```

VCVTUDQ2PS __m512 __mm512_cvtepu32_ps( __m512i a);
VCVTUDQ2PS __m512 __mm512_mask_cvtepu32_ps( __m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_maskz_cvtepu32_ps( __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 __mm512_cvt_roundepu32_ps( __m512i a, int r);
VCVTUDQ2PS __m512 __mm512_mask_cvt_roundepu32_ps( __m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 __mm512_maskz_cvt_roundepu32_ps( __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 __mm256_cvtepu32_ps( __m256i a);
VCVTUDQ2PS __m256 __mm256_mask_cvtepu32_ps( __m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 __mm256_maskz_cvtepu32_ps( __mmask8 k, __m256i a);
VCVTUDQ2PS __m128 __mm_cvtepu32_ps( __m128i a);
VCVTUDQ2PS __m128 __mm_mask_cvtepu32_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 __mm_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

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| EVEX.128.F3.0F.W1 7A /r<br>VCVTUQQ2PD xmm1 {k1}{z},<br>xmm2/m128/m64bcst     | A          | V/V                          | AVX512VL<br>AVX512DQ     | Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double precision floating-point values in xmm1 with writemask k1.     |
| EVEX.256.F3.0F.W1 7A /r<br>VCVTUQQ2PD ymm1 {k1}{z},<br>ymm2/m256/m64bcst     | A          | V/V                          | AVX512VL<br>AVX512DQ     | Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double precision floating-point values in ymm1 with writemask k1.        |
| EVEX.512.F3.0F.W1 7A /r<br>VCVTUQQ2PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{er} | A          | V/V                          | AVX512DQ                 | Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double precision floating-point values in zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

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

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

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

### Operation

#### VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Register

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 64

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

THEN DEST[i+63:i] :=

Convert\_UQuadInteger\_To\_Double\_Precision\_Floating\_Point(SRC[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VCVTUQQ2PD (EVEX Encoded Version) When SRC Operand is a Memory Source**

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

```

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

```

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

```

VCVTUQQ2PD __m512d _mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PD __m512d _mm512_mask_cvtepu64_ps( __m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_mask_cvt_roundepu64_ps( __m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d _mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PD __m256d _mm256_mask_cvtepu64_ps( __m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PD __m128d _mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PD __m128d _mm_mask_cvtepu64_ps( __m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d _mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

#UD If EVEX.vvvv != 1111B.

## VCVTUQQ2PH—Convert Packed Unsigned Quadword Integers to Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.F2.MAP5.W1 7A /r<br>VCVTUQQ2PH xmm1{k1}{z}, xmm2/<br>m128/m64bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert two packed unsigned doubleword integers from xmm2/m128/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.   |
| EVEX.256.F2.MAP5.W1 7A /r<br>VCVTUQQ2PH xmm1{k1}{z}, ymm2/<br>m256/m64bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert four packed unsigned doubleword integers from ymm2/m256/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1.  |
| EVEX.512.F2.MAP5.W1 7A /r<br>VCVTUQQ2PH xmm1{k1}{z}, zmm2/<br>m512/m64bcst {er} | A         | V/V                          | AVX512-FP16             | Convert eight packed unsigned doubleword integers from zmm2/m512/m64bcst to packed FP16 values, and store the result in xmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

#### Description

This instruction converts packed unsigned quadword integers in the source operand to packed FP16 values in the destination operand. The destination elements are updated according to the writemask.

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

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

#### Operation

**VCVTUQQ2PH dest, src**

VL = 128, 256 or 512

KL := VL / 64

IF \*SRC is a register\* and (VL = 512) AND (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.qword[0]

ELSE

tsrc := SRC.qword[j]

DEST.fp16[j] := Convert\_unsigned\_integer64\_to\_fp16(tsrc)

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL/4] := 0

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

VCVTUQQ2PH \_\_m128h \_\_mm512\_cvt\_roundepu64\_ph (\_\_m512i a, int rounding);  
 VCVTUQQ2PH \_\_m128h \_\_mm512\_mask\_cvt\_roundepu64\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m512i a, int rounding);  
 VCVTUQQ2PH \_\_m128h \_\_mm512\_maskz\_cvt\_roundepu64\_ph (\_\_mmask8 k, \_\_m512i a, int rounding);  
 VCVTUQQ2PH \_\_m128h \_\_mm\_cvtepu64\_ph (\_\_m128i a);  
 VCVTUQQ2PH \_\_m128h \_\_mm\_mask\_cvtepu64\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128i a);  
 VCVTUQQ2PH \_\_m128h \_\_mm\_maskz\_cvtepu64\_ph (\_\_mmask8 k, \_\_m128i a);  
 VCVTUQQ2PH \_\_m128h \_\_mm256\_cvtepu64\_ph (\_\_m256i a);  
 VCVTUQQ2PH \_\_m128h \_\_mm256\_mask\_cvtepu64\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m256i a);  
 VCVTUQQ2PH \_\_m128h \_\_mm256\_maskz\_cvtepu64\_ph (\_\_mmask8 k, \_\_m256i a);  
 VCVTUQQ2PH \_\_m128h \_\_mm512\_cvtepu64\_ph (\_\_m512i a);  
 VCVTUQQ2PH \_\_m128h \_\_mm512\_mask\_cvtepu64\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m512i a);  
 VCVTUQQ2PH \_\_m128h \_\_mm512\_maskz\_cvtepu64\_ph (\_\_mmask8 k, \_\_m512i a);

**SIMD Floating-Point Exceptions**

Overflow, Precision.

**Other Exceptions**

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

## VCVTUSI2SH—Convert Unsigned Doubleword Integer to an FP16 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F3.MAP5.W0 7B /r<br>VCVTUSI2SH xmm1, xmm2, r32/m32<br>{er} | A         | V/V <sup>1</sup>             | AVX512-FP16           | Convert an unsigned doubleword integer from r32/m32 to an FP16 value, and store the result in xmm1. Bits 127:16 from xmm2 are copied to xmm1[127:16]. |
| EVEX.LLIG.F3.MAP5.W1 7B /r<br>VCVTUSI2SH xmm1, xmm2, r64/m64<br>{er} | A         | V/N.E.                       | AVX512-FP16           | Convert an unsigned quadword integer from r64/m64 to an FP16 value, and store the result in xmm1. Bits 127:16 from xmm2 are copied to xmm1[127:16].   |

### NOTES:

1. Outside of 64b mode, the EVEX.W field is ignored. The instruction behaves as if W=0 was used.

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a FP16 value in the destination operand. The result is stored in the low word of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

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 127:16 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.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

### Operation

**VCVTUSI2SH dest, src1, src2**

IF \*SRC2 is a register\* and (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

IF 64-mode and OperandSize == 64:

DEST.fp16[0] := Convert\_unsigned\_integer64\_to\_fp16(SRC2.qword)

ELSE:

DEST.fp16[0] := Convert\_unsigned\_integer32\_to\_fp16(SRC2.dword)

DEST[127:16] := SRC1[127:16]

DEST[MAXVL-1:128] := 0

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

VCVTUSI2SH \_\_m128h \_mm\_cvt\_roundu32\_sh (\_\_m128h a, unsigned int b, int rounding);  
VCVTUSI2SH \_\_m128h \_mm\_cvt\_roundu64\_sh (\_\_m128h a, unsigned \_\_int64 b, int rounding);  
VCVTUSI2SH \_\_m128h \_mm\_cvtu32\_sh (\_\_m128h a, unsigned int b);  
VCVTUSI2SH \_\_m128h \_mm\_cvtu64\_sh (\_\_m128h a, unsigned \_\_int64 b);

**SIMD Floating-Point Exceptions**

Overflow, Precision.

**Other Exceptions**

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

## VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single Precision Floating-Point Values

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

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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

FI;

ENDFOR

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



**VCVTUQQ2PS (EVEX Encoded Version) When SRC Operand is a Memory Source**

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

```

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

```

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

```

VCVTUQQ2PS __m256 __mm512_cvtepu64_ps( __m512i a);
VCVTUQQ2PS __m256 __mm512_mask_cvtepu64_ps( __m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __mm512_maskz_cvtepu64_ps( __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 __mm512_cvt_roundepu64_ps( __m512i a, int r);
VCVTUQQ2PS __m256 __mm512_mask_cvt_roundepu64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 __mm512_maskz_cvt_roundepu64_ps( __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128 __mm256_cvtepu64_ps( __m256i a);
VCVTUQQ2PS __m128 __mm256_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 __mm256_maskz_cvtepu64_ps( __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 __mm_cvtepu64_ps( __m128i a);
VCVTUQQ2PS __m128 __mm_mask_cvtepu64_ps( __m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 __mm_maskz_cvtepu64_ps( __mmask8 k, __m128i a);

```

**SIMD Floating-Point Exceptions**

Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

Additionally:

#UD If EVEX.vvvv != 1111B.

## VCVTUSI2SD—Convert Unsigned Integer to Scalar Double Precision Floating-Point Value

| Opcode/<br>Instruction                                       | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.F2.OF.W0 7B /r<br>VCVTUSI2SD xmm1, xmm2, r/m32     | A         | V/V                          | AVX512F                  | Convert one unsigned doubleword integer from r/m32 to one double precision floating-point value in xmm1. |
| EVEX.LLIG.F2.OF.W1 7B /r<br>VCVTUSI2SD xmm1, xmm2, r/m64{er} | A         | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one unsigned quadword integer from r/m64 to one double precision floating-point value in xmm1.   |

### NOTES:

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

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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\_cvtsd( \_\_m128d s, unsigned a);

VCVTUSI2SD \_\_m128d \_mm\_cvtsd64( \_\_m128d s, unsigned \_\_int64 a);

VCVTUSI2SD \_\_m128d \_mm\_cvt\_roundsd( \_\_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; otherwise, see Table 2-59, “Type E10NF Class Exception Conditions.”

## VCVTUSI2SS—Convert Unsigned Integer to Scalar Single Precision Floating-Point Value

| Opcode/<br>Instruction                                       | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.F3.OF.W0 7B /r<br>VCVTUSI2SS xmm1, xmm2, r/m32{er} | A         | V/V                          | AVX512F                  | Convert one signed doubleword integer from r/m32 to one single precision floating-point value in xmm1. |
| EVEX.LLIG.F3.OF.W1 7B /r<br>VCVTUSI2SS xmm1, xmm2, r/m64{er} | A         | V/N.E. <sup>1</sup>          | AVX512F                  | Convert one signed quadword integer from r/m64 to one single precision floating-point value in xmm1.   |

### NOTES:

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

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|---------------|---------------|--------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

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

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

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

### Operation

#### VCVTUSI2SS (EVEX Encoded Version)

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] := Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] := Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[127:32] := SRC1[127:32]

DEST[MAXVL-1:128] := 0

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

VCVTUSI2SS \_\_m128 \_\_mm\_cvtsi32\_ss( \_\_m128 s, unsigned a);

VCVTUSI2SS \_\_m128 \_\_mm\_cvt\_roundsi32\_ss( \_\_m128 s, unsigned a, int r);

VCVTUSI2SS \_\_m128 \_\_mm\_cvtsi64\_ss( \_\_m128 s, unsigned \_\_int64 a);

VCVTUSI2SS \_\_m128 \_\_mm\_cvt\_roundsi64\_ss( \_\_m128 s, unsigned \_\_int64 a, int r);

**SIMD Floating-Point Exceptions**

Precision.

**Other Exceptions**

See Table 2-48, “Type E3NF Class Exception Conditions.”

## VCVTUW2PH—Convert Packed Unsigned Word Integers to FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.F2.MAP5.W0 7D /r<br>VCVTUW2PH xmm1{k1}{z}, xmm2/<br>m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed unsigned word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.      |
| EVEX.256.F2.MAP5.W0 7D /r<br>VCVTUW2PH ymm1{k1}{z}, ymm2/<br>m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert sixteen packed unsigned word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.    |
| EVEX.512.F2.MAP5.W0 7D /r<br>VCVTUW2PH zmm1{k1}{z}, zmm2/<br>m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Convert thirty-two packed unsigned word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed unsigned word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

If the result of the convert operation is overflow and MXCSR.OM=0 then a SIMD exception will be raised with OE=1, PE=1.

### Operation

**VCVTUW2PH dest, src**

VL = 128, 256 or 512

KL := VL / 16

IF \*SRC is a register\* and (VL = 512) AND (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.word[0]

ELSE

tsrc := SRC.word[j]

DEST.fp16[j] := Convert\_unsigned\_integer16\_to\_fp16(tsrc)

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VCVTUW2PH \_\_m512h \_mm512\_cvt\_roundedu16\_ph (\_\_m512i a, int rounding);  
 VCVTUW2PH \_\_m512h \_mm512\_mask\_cvt\_roundedu16\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512i a, int rounding);  
 VCVTUW2PH \_\_m512h \_mm512\_maskz\_cvt\_roundedu16\_ph (\_\_mmask32 k, \_\_m512i a, int rounding);  
 VCVTUW2PH \_\_m128h \_mm\_cvtepu16\_ph (\_\_m128i a);  
 VCVTUW2PH \_\_m128h \_mm\_mask\_cvtepu16\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128i a);  
 VCVTUW2PH \_\_m128h \_mm\_maskz\_cvtepu16\_ph (\_\_mmask8 k, \_\_m128i a);  
 VCVTUW2PH \_\_m256h \_mm256\_cvtepu16\_ph (\_\_m256i a);  
 VCVTUW2PH \_\_m256h \_mm256\_mask\_cvtepu16\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256i a);  
 VCVTUW2PH \_\_m256h \_mm256\_maskz\_cvtepu16\_ph (\_\_mmask16 k, \_\_m256i a);  
 VCVTUW2PH \_\_m512h \_mm512\_cvtepu16\_ph (\_\_m512i a);  
 VCVTUW2PH \_\_m512h \_mm512\_mask\_cvtepu16\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512i a);  
 VCVTUW2PH \_\_m512h \_mm512\_maskz\_cvtepu16\_ph (\_\_mmask32 k, \_\_m512i a);

**SIMD Floating-Point Exceptions**

Overflow, Precision.

**Other Exceptions**

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

## VCVTW2PH—Convert Packed Signed Word Integers to FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.F3.MAP5.W0 7D /r<br>VCVTW2PH xmm1{k1}{z}, xmm2/<br>m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert eight packed signed word integers from xmm2/m128/m16bcst to FP16 values, and store the result in xmm1 subject to writemask k1.      |
| EVEX.256.F3.MAP5.W0 7D /r<br>VCVTW2PH ymm1{k1}{z}, ymm2/<br>m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert sixteen packed signed word integers from ymm2/m256/m16bcst to FP16 values, and store the result in ymm1 subject to writemask k1.    |
| EVEX.512.F3.MAP5.W0 7D /r<br>VCVTW2PH zmm1{k1}{z}, zmm2/<br>m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Convert thirty-two packed signed word integers from zmm2/m512/m16bcst to FP16 values, and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction converts packed signed word integers in the source operand to FP16 values in the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or embedded rounding controls.

The destination elements are updated according to the writemask.

### Operation

**VCVTW2PH dest, src**

VL = 128, 256 or 512

KL := VL / 16

IF \*SRC is a register\* and (VL = 512) AND (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE:

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*SRC is memory\* and EVEX.b = 1:

tsrc := SRC.word[0]

ELSE

tsrc := SRC.word[j]

DEST.fp16[j] := Convert\_integer16\_to\_fp16(tsrc)

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0



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

```

VCVTW2PH __m512h __mm512_cvt_roundepi16_ph (__m512i a, int rounding);
VCVTW2PH __m512h __mm512_mask_cvt_roundepi16_ph (__m512h src, __mmask32 k, __m512i a, int rounding);
VCVTW2PH __m512h __mm512_maskz_cvt_roundepi16_ph (__mmask32 k, __m512i a, int rounding);
VCVTW2PH __m128h __mm_cvtepi16_ph (__m128i a);
VCVTW2PH __m128h __mm_mask_cvtepi16_ph (__m128h src, __mmask8 k, __m128i a);
VCVTW2PH __m128h __mm_maskz_cvtepi16_ph (__mmask8 k, __m128i a);
VCVTW2PH __m256h __mm256_cvtepi16_ph (__m256i a);
VCVTW2PH __m256h __mm256_mask_cvtepi16_ph (__m256h src, __mmask16 k, __m256i a);
VCVTW2PH __m256h __mm256_maskz_cvtepi16_ph (__mmask16 k, __m256i a);
VCVTW2PH __m512h __mm512_cvtepi16_ph (__m512i a);
VCVTW2PH __m512h __mm512_mask_cvtepi16_ph (__m512h src, __mmask32 k, __m512i a);
VCVTW2PH __m512h __mm512_maskz_cvtepi16_ph (__mmask32 k, __m512i a);

```

**SIMD Floating-Point Exceptions**

Precision.

**Other Exceptions**

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

## VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W0 42 /r ib<br>VDBPSADBW xmm1 {k1}{z}, xmm2,<br>xmm3/m128, imm8 | A         | V/V                          | AVX512VL<br>AVX512BW     | 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. |
| EVEX.256.66.0F3A.W0 42 /r ib<br>VDBPSADBW ymm1 {k1}{z}, ymm2,<br>ymm3/m256, imm8 | A         | V/V                          | AVX512VL<br>AVX512BW     | 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. |
| EVEX.512.66.0F3A.W0 42 /r ib<br>VDBPSADBW zmm1 {k1}{z}, zmm2,<br>zmm3/m512, imm8 | A         | V/V                          | AVX512BW                 | 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. |

### Instruction Operand Encoding

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

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

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.

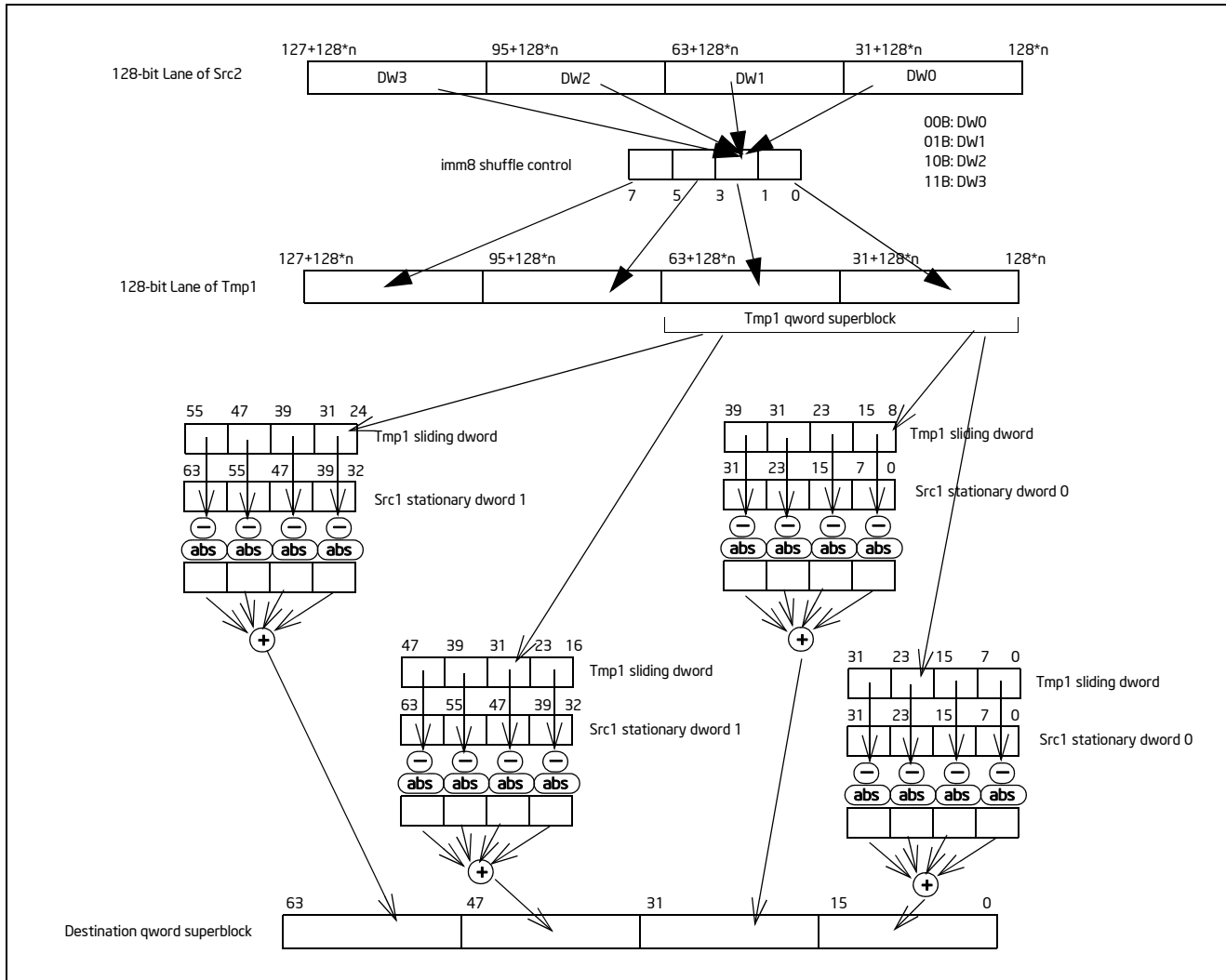


Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBw

## Operation

### VDBPSADBw (EVEX Encoded Versions)

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

Selection of quadruplets:

FOR I = 0 to VL step 128

TMP1[ $I+31:I$ ] := select (SRC2[ $I+127:I$ ], imm8[1:0])

TMP1[ $I+63:I+32$ ] := select (SRC2[ $I+127:I$ ], imm8[3:2])

TMP1[ $I+95:I+64$ ] := select (SRC2[ $I+127:I$ ], imm8[5:4])

TMP1[ $I+127:I+96$ ] := select (SRC2[ $I+127:I$ ], imm8[7:6])

END FOR

SAD of quadruplets:

FOR I = 0 to VL step 64

TMP\_DEST[ $I+15:I$ ] := ABS(SRC1[ $I+7:I$ ] - TMP1[ $I+7:I$ ]) +  
ABS(SRC1[ $I+15:I+8$ ] - TMP1[ $I+15:I+8$ ]) +

```
ABS(SRC1[I+23: I+16]- TMP1[I+23: I+16]) +
ABS(SRC1[I+31: I+24]- TMP1[I+31: I+24])
```

```
TMP_DEST[I+31: I+16] := ABS(SRC1[I+7: I] - TMP1[I+15: I+8]) +
ABS(SRC1[I+15: I+8]- TMP1[I+23: I+16]) +
ABS(SRC1[I+23: I+16]- TMP1[I+31: I+24]) +
ABS(SRC1[I+31: I+24]- TMP1[I+39: I+32])
TMP_DEST[I+47: I+32] := ABS(SRC1[I+39: I+32] - TMP1[I+23: I+16]) +
ABS(SRC1[I+47: I+40]- TMP1[I+31: I+24]) +
ABS(SRC1[I+55: I+48]- TMP1[I+39: I+32]) +
ABS(SRC1[I+63: I+56]- TMP1[I+47: I+40])
```

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

```
ENDFOR
```

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

```
  i := j * 16
```

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

```
    THEN DEST[i+15:i] := TMP_DEST[i+15:i]
```

```
  ELSE
```

```
    IF *merging-masking* ; merging-masking
```

```
      THEN *DEST[i+15:i] remains unchanged*
```

```
    ELSE ; zeroing-masking
```

```
      DEST[i+15:i] := 0
```

```
    FI
```

```
  FI;
```

```
ENDFOR
```

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

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

```
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."

## VDIVPH—Divide Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.MAP5.WO 5E /r<br>VDIVPH xmm1{k1}{z}, xmm2, xmm3/<br>m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Divide packed FP16 values in xmm2 by packed FP16 values in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1. |
| EVEX.256.NP.MAP5.WO 5E /r<br>VDIVPH ymm1{k1}{z}, ymm2, ymm3/<br>m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Divide packed FP16 values in ymm2 by packed FP16 values in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1. |
| EVEX.512.NP.MAP5.WO 5E /r<br>VDIVPH zmm1{k1}{z}, zmm2, zmm3/<br>m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Divide packed FP16 values in zmm2 by packed FP16 values in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|---------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction divides packed FP16 values from the first source operand by the corresponding elements in the second source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

### Operation

#### VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Register

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VDIVPH (EVEX Encoded Versions) When SRC2 Operand is a Memory Source**

VL = 128, 256 or 512

KL := VL/16

```

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF EVEX.b = 1:
      DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[0]
    ELSE:
      DEST.fp16[j] := SRC1.fp16[j] / SRC2.fp16[j]
  ELSE IF *zeroing*:
    DEST.fp16[j] := 0
  // else dest.fp16[j] remains unchanged

```

DEST[MAXVL-1:VL] := 0

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

```

VDIVPH __m128h _mm_div_ph (__m128h a, __m128h b);
VDIVPH __m128h _mm_mask_div_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VDIVPH __m128h _mm_maskz_div_ph (__mmask8 k, __m128h a, __m128h b);
VDIVPH __m256h _mm256_div_ph (__m256h a, __m256h b);
VDIVPH __m256h _mm256_mask_div_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VDIVPH __m256h _mm256_maskz_div_ph (__mmask16 k, __m256h a, __m256h b);
VDIVPH __m512h _mm512_div_ph (__m512h a, __m512h b);
VDIVPH __m512h _mm512_mask_div_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_maskz_div_ph (__mmask32 k, __m512h a, __m512h b);
VDIVPH __m512h _mm512_div_round_ph (__m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_mask_div_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int rounding);
VDIVPH __m512h _mm512_maskz_div_round_ph (__mmask32 k, __m512h a, __m512h b, int rounding);

```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

## VDIVSH—Divide Scalar FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|---|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.F3.MAP5.WO 5E /r<br>VDIVSH xmm1{k1}{z}, xmm2, xmm3/<br>m16 {er} | A         | V/V                          | AVX512-FP16           | Divide low FP16 value in xmm2 by low FP16 value in xmm3/m16, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction divides the low FP16 value from the first source operand by the corresponding value in the second source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

#### VDIVSH (EVEX Encoded Versions)

IF EVEX.b = 1 and SRC2 is a register:

```
SET_RM(EVEX.RC)
```

ELSE

```
SET_RM(MXCSR.RC)
```

IF k1[0] OR \*no writemask\*:

```
DEST.fp16[0] := SRC1.fp16[0] / SRC2.fp16[0]
```

ELSE IF \*zeroing\*:

```
DEST.fp16[0] := 0
```

// else dest.fp16[0] remains unchanged

```
DEST[127:16] := SRC1[127:16]
```

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

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

```
VDIVSH __m128h __mm_div_round_sh (__m128h a, __m128h b, int rounding);
```

```
VDIVSH __m128h __mm_mask_div_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
```

```
VDIVSH __m128h __mm_maskz_div_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
```

```
VDIVSH __m128h __mm_div_sh (__m128h a, __m128h b);
```

```
VDIVSH __m128h __mm_mask_div_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
```

```
VDIVSH __m128h __mm_maskz_div_sh (__mmask8 k, __m128h a, __m128h b);
```

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal, Zero.

### Other Exceptions

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

## VDPBF16PS—Dot Product of BF16 Pairs Accumulated Into Packed Single Precision

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.F3.0F38.W0 52 /r<br>VDPBF16PS xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst | A         | V/V                          | AVX512VL<br>AVX512_BF16 | Multiply BF16 pairs from xmm2 and<br>xmm3/m128, and accumulate the resulting<br>packed single precision results in xmm1 with<br>writemask k1. |
| EVEX.256.F3.0F38.W0 52 /r<br>VDPBF16PS ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst | A         | V/V                          | AVX512VL<br>AVX512_BF16 | Multiply BF16 pairs from ymm2 and<br>ymm3/m256, and accumulate the resulting<br>packed single precision results in ymm1 with<br>writemask k1. |
| EVEX.512.F3.0F38.W0 52 /r<br>VDPBF16PS zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst | A         | V/V                          | AVX512F<br>AVX512_BF16  | Multiply BF16 pairs from zmm2 and<br>zmm3/m512, and accumulate the resulting<br>packed single precision results in zmm1 with<br>writemask k1. |

### Instruction Operand Encoding

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

### 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**

| NaN Priority | Description      | Comments  |
|--------------|------------------|---|
| 1            | src1 low is NaN  | Lower part has priority over upper part, i.e., it overrides the upper part. |
| 2            | src2 low is NaN  |   |
| 3            | src1 high is NaN | Upper part may be overridden if lower has NaN.                              |
| 4            | src2 high is NaN |   |
| 5            | srcdest is NaN   | Dest is propagated if no NaN is encountered by src2.                        |

### 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.bfloat[1])

srcdest.fp32[ i ] += make\_fp32(src1.bfloat16[2\*i+0]) \* make\_fp32(t.bfloat[0])

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."

## VERR/VERW—Verify a Segment for Reading or Writing

| Opcode/<br>Instruction | Op/<br>En | 64-Bit<br>Mode | Compat/<br>Leg<br>Mode | Description  |
|------------------------|-----------|----------------|------------------------|--|
| OF 00 /4<br>VERR r/m16 | M         | Valid          | Valid                  | Set ZF=1 if segment specified with r/m16 can be read.    |
| OF 00 /5<br>VERW r/m16 | M         | Valid          | Valid                  | Set ZF=1 if segment specified with r/m16 can be written. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2 | Operand 3 | Operand 4 |
|-------|---------------|-----------|-----------|-----------|
| M     | ModRM:r/m (r) | N/A       | N/A       | N/A       |

### 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; FI;
```

Read segment descriptor;

```
IF SegmentDescriptor(DescriptorType) = 0 (* System segment *)
or (SegmentDescriptor(Type) ≠ conforming code segment)
and (CPL > DPL) or (RPL > DPL)
  THEN
    ZF := 0;
  ELSE
    IF ((Instruction = VERR) and (Segment readable))
    or ((Instruction = VERW) and (Segment writable))
      THEN
        ZF := 1;
      ELSE
        ZF := 0;
```

FI;  
FI;

### 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.<br>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.<br>If the LOCK prefix is used. |
|-----|--|

### Virtual-8086 Mode Exceptions

|     |  |
|-----|--|
| #UD | The VERR and VERW instructions are not recognized in virtual-8086 mode.<br>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.  |

## VEXPANDPD—Load Sparse Packed Double Precision Floating-Point Values From Dense Memory

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 88 /r<br>VEXPANDPD xmm1 {k1}{z},<br>xmm2/m128 | A         | V/V                          | AVX512VL<br>AVX512F      | Expand packed double precision floating-point values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 88 /r<br>VEXPANDPD ymm1 {k1}{z}, ymm2/m256    | A         | V/V                          | AVX512VL<br>AVX512F      | Expand packed double precision floating-point values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.66.0F38.W1 88 /r<br>VEXPANDPD zmm1 {k1}{z}, zmm2/m512    | A         | V/V                          | AVX512F                  | Expand packed double precision floating-point values from zmm2/m512 to zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

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

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

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

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

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

### Operation

#### VEXPANDPD (EVEX Encoded Versions)

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

k := 0

FOR j := 0 TO KL-1

  i := j \* 64

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+63:i] := SRC[k+63:k];

      k := k + 64

    ELSE

      IF \*merging-masking\* ; merging-masking

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

      ELSE ; zeroing-masking

        THEN DEST[i+63:i] := 0

    FI

  FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

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

```

VEXPANDPD __m512d __mm512_mask_expand_pd( __m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d __mm512_maskz_expand_pd( __mmask8 k, __m512d a);
VEXPANDPD __m512d __mm512_mask_expandloadu_pd( __m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d __mm512_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m256d __mm256_mask_expand_pd( __m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d __mm256_maskz_expand_pd( __mmask8 k, __m256d a);
VEXPANDPD __m256d __mm256_mask_expandloadu_pd( __m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d __mm256_maskz_expandloadu_pd( __mmask8 k, void * a);
VEXPANDPD __m128d __mm_mask_expand_pd( __m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d __mm_maskz_expand_pd( __mmask8 k, __m128d a);
VEXPANDPD __m128d __mm_mask_expandloadu_pd( __m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d __mm_maskz_expandloadu_pd( __mmask8 k, void * a);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4.nb 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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 88 /r<br>VEXPANDPS xmm1 {k1}{z}, xmm2/m128 | A         | V/V                          | AVX512VL<br>AVX512F      | Expand packed single precision floating-point values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 88 /r<br>VEXPANDPS ymm1 {k1}{z}, ymm2/m256 | A         | V/V                          | AVX512VL<br>AVX512F      | Expand packed single precision floating-point values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 88 /r<br>VEXPANDPS zmm1 {k1}{z}, zmm2/m512 | A         | V/V                          | AVX512F                  | Expand packed single precision floating-point values from zmm2/m512 to zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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

FOR j := 0 TO KL-1

  i := j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+31:i] := SRC[k+31:k];

      k := k + 32

    ELSE

      IF \*merging-masking\* ; merging-masking

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

        ELSE ; zeroing-masking

          DEST[i+31:i] := 0

    FI

  FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

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

```

VEXPANDPS __m512 __mm512_mask_expand_ps( __m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 __mm512_maskz_expand_ps( __mmask16 k, __m512 a);
VEXPANDPS __m512 __mm512_mask_expandloadu_ps( __m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 __mm512_maskz_expandloadu_ps( __mmask16 k, void * a);
VEXPANDPD __m256 __mm256_mask_expand_ps( __m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 __mm256_maskz_expand_ps( __mmask8 k, __m256 a);
VEXPANDPD __m256 __mm256_mask_expandloadu_ps( __m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 __mm256_maskz_expandloadu_ps( __mmask8 k, void * a);
VEXPANDPD __m128 __mm_mask_expand_ps( __m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 __mm_maskz_expand_ps( __mmask8 k, __m128 a);
VEXPANDPD __m128 __mm_mask_expandloadu_ps( __m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 __mm_maskz_expandloadu_ps( __mmask8 k, void * a);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."

Additionally:

#UD                    If EVEX.vvvv != 1111B.

## VEEXTRACTF128/VEEXTRACTF32x4/VEEXTRACTF64x2/VEEXTRACTF32x8/VEEXTRACTF64x4— Extract Packed Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.256.66.0F3A.W0 19 /r ib<br>VEEXTRACTF128 xmm1/m128, ymm2,<br>imm8           | A         | V/V                          | AVX                      | Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.  |
| EVEX.256.66.0F3A.W0 19 /r ib<br>VEEXTRACTF32X4 xmm1/m128 {k1}{z},<br>ymm2, imm8 | C         | V/V                          | AVX512VL<br>AVX512F      | Extract 128 bits of packed single precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.512.66.0F3A.W0 19 /r ib<br>VEEXTRACTF32x4 xmm1/m128 {k1}{z},<br>zmm2, imm8 | C         | V/V                          | AVX512F                  | Extract 128 bits of packed single precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.256.66.0F3A.W1 19 /r ib<br>VEEXTRACTF64X2 xmm1/m128 {k1}{z},<br>ymm2, imm8 | B         | V/V                          | AVX512VL<br>AVX512DQ     | Extract 128 bits of packed double precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.512.66.0F3A.W1 19 /r ib<br>VEEXTRACTF64X2 xmm1/m128 {k1}{z},<br>zmm2, imm8 | B         | V/V                          | AVX512DQ                 | Extract 128 bits of packed double precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.512.66.0F3A.W0 1B /r ib<br>VEEXTRACTF32X8 ymm1/m256 {k1}{z},<br>zmm2, imm8 | D         | V/V                          | AVX512DQ                 | Extract 256 bits of packed single precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1. |
| EVEX.512.66.0F3A.W1 1B /r ib<br>VEEXTRACTF64x4 ymm1/m256 {k1}{z},<br>zmm2, imm8 | C         | V/V                          | AVX512F                  | Extract 256 bits of packed double precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | N/A        | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |
| B     | Tuple2     | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |
| C     | Tuple4     | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |
| D     | Tuple8     | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |

### Description

VEEXTRACTF128/VEEXTRACTF32x4 and VEEXTRACTF64x2 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.

VEEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEEXTRACTF32x8 and VEEXTRACTF64x4 extract 256-bits of double precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

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

The high 6 bits of the immediate are ignored.



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

### Operation

#### VEXTRACTF32x4 (EVEX Encoded Versions) When Destination is a Register

VL = 256, 512

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] := SRC1[127:0]

1: TMP\_DEST[127:0] := SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] := SRC1[127:0]

01: TMP\_DEST[127:0] := SRC1[255:128]

10: TMP\_DEST[127:0] := SRC1[383:256]

11: TMP\_DEST[127:0] := SRC1[511:384]

ESAC.

FI;

FOR j := 0 TO 3

i := j \* 32

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

THEN DEST[i+31:i] := TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] := 0

FI

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]

1: TMP\_DEST[127:0] := SRC1[255:128]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] := SRC1[127:0]

01: TMP\_DEST[127:0] := SRC1[255:128]

10: TMP\_DEST[127:0] := SRC1[383:256]

11: TMP\_DEST[127:0] := SRC1[511:384]

ESAC.

FI;

FOR j := 0 TO 3

i := j \* 32

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

```

        THEN DEST[i+31:i] := TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR

```

**VEXTRACTF64x2 (EVEX Encoded Versions) When Destination is a Register**

VL = 256, 512

IF VL = 256

```

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

```

```

FI;
IF VL = 512

```

```

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

```

```

FI;

FOR j := 0 TO 1

```

```

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

```

```

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]
        1: TMP_DEST[127:0] := SRC1[255:128]
    ESAC.

```

```

FI;
IF VL = 512

```

```

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

```

```

FI;

FOR j := 0 TO 1

```

```

i := j * 64
IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE *DEST[i+63:i] remains unchanged* ; merging-masking
FI;
ENDFOR

```

#### **VEEXTRACTF32x8 (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

```

#### **VEEXTRACTF32x8 (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

```

#### **VEEXTRACTF64x4 (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[j+63:i] remains unchanged*
            ELSE *zeroing-masking*     ; zeroing-masking
                DEST[j+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:256] := 0

```

**VEEXTRACTF64x4 (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[j+63:i] := TMP_DEST[j+63:i]
        ELSE           ; merging-masking
            *DEST[j+63:i] remains unchanged*
    FI;
ENDFOR

```

**VEEXTRACTF128 (Memory Destination Form)**

```

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

```

**VEEXTRACTF128 (Register Destination Form)**

```

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

```

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

```

VEEXTRACTF32x4 __m128 __mm512_extractf32x4_ps(__m512 a, const int nid);
VEEXTRACTF32x4 __m128 __mm512_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m512 a, const int nid);
VEEXTRACTF32x4 __m128 __mm512_maskz_extractf32x4_ps( __mmask8 k, __m512 a, const int nid);
VEEXTRACTF32x4 __m128 __mm256_extractf32x4_ps(__m256 a, const int nid);
VEEXTRACTF32x4 __m128 __mm256_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m256 a, const int nid);
VEEXTRACTF32x4 __m128 __mm256_maskz_extractf32x4_ps( __mmask8 k, __m256 a, const int nid);
VEEXTRACTF32x8 __m256 __mm512_extractf32x8_ps(__m512 a, const int nid);
VEEXTRACTF32x8 __m256 __mm512_mask_extractf32x8_ps(__m256 s, __mmask8 k, __m512 a, const int nid);
VEEXTRACTF32x8 __m256 __mm512_maskz_extractf32x8_ps( __mmask8 k, __m512 a, const int nid);
VEEXTRACTF64x2 __m128d __mm512_extractf64x2_pd(__m512d a, const int nid);
VEEXTRACTF64x2 __m128d __mm512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nid);
VEEXTRACTF64x2 __m128d __mm512_maskz_extractf64x2_pd( __mmask8 k, __m512d a, const int nid);
VEEXTRACTF64x2 __m128d __mm256_extractf64x2_pd(__m256d a, const int nid);
VEEXTRACTF64x2 __m128d __mm256_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m256d a, const int nid);
VEEXTRACTF64x2 __m128d __mm256_maskz_extractf64x2_pd( __mmask8 k, __m256d a, const int nid);
VEEXTRACTF64x4 __m256d __mm512_extractf64x4_pd( __m512d a, const int nid);

```

VEXTRACTF64x4 \_\_m256d \_\_mm512\_mask\_extractf64x4\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m512d a, const int nidx);  
 VEXTRACTF64x4 \_\_m256d \_\_mm512\_maskz\_extractf64x4\_pd(\_\_mmask8 k, \_\_m512d a, const int nidx);  
 VEXTRACTF128 \_\_m128 \_\_mm256\_extractf128\_ps(\_\_m256 a, int offset);  
 VEXTRACTF128 \_\_m128d \_\_mm256\_extractf128\_pd(\_\_m256d a, int offset);  
 VEXTRACTF128 \_\_m128i \_\_mm256\_extractf128\_si256(\_\_m256i a, int offset);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

VEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-54, “Type E6NF Class Exception Conditions.”

Additionally:

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VEEXTRACTI128/VEEXTRACTI32x4/VEEXTRACTI64x2/VEEXTRACTI32x8/VEEXTRACTI64x4—Extract Packed Integer Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| VEEX.256.66.0F3A.W0 39 /r ib<br>VEEXTRACTI128 xmm1/m128, ymm2,<br>imm8          | A         | V/V                          | AVX2                     | Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.                                       |
| EVEX.256.66.0F3A.W0 39 /r ib<br>VEEXTRACTI32X4 xmm1/m128 {k1}{z},<br>ymm2, imm8 | C         | V/V                          | AVX512VL<br>AVX512F      | Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.512.66.0F3A.W0 39 /r ib<br>VEEXTRACTI32x4 xmm1/m128 {k1}{z},<br>zmm2, imm8 | C         | V/V                          | AVX512F                  | Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1. |
| EVEX.256.66.0F3A.W1 39 /r ib<br>VEEXTRACTI64X2 xmm1/m128 {k1}{z},<br>ymm2, imm8 | B         | V/V                          | AVX512VL<br>AVX512DQ     | Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.   |
| EVEX.512.66.0F3A.W1 39 /r ib<br>VEEXTRACTI64X2 xmm1/m128 {k1}{z},<br>zmm2, imm8 | B         | V/V                          | AVX512DQ                 | Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.   |
| EVEX.512.66.0F3A.W0 3B /r ib<br>VEEXTRACTI32X8 ymm1/m256 {k1}{z},<br>zmm2, imm8 | D         | V/V                          | AVX512DQ                 | Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1. |
| EVEX.512.66.0F3A.W1 3B /r ib<br>VEEXTRACTI64x4 ymm1/m256 {k1}{z},<br>zmm2, imm8 | C         | V/V                          | AVX512F                  | Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.   |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | N/A        | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |
| B     | Tuple2     | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |
| C     | Tuple4     | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |
| D     | Tuple8     | ModRM:r/m (w) | ModRM:reg (r) | imm8      | N/A       |

#### Description

VEEXTRACTI128/VEEXTRACTI32x4 and VEEXTRACTI64x2 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.

VEEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEEXTRACTI32x8 and VEEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor. The destination may be either a vector register or a 256-bit memory location.

VEEXTRACTI32x8: The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.

**VEEXTRACTI64x4:** The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEEX.vvvv and EVEEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits (6 bits in EVEEX.512) of the immediate are ignored.

If VEEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

### Operation

#### **VEEXTRACTI32x4 (EVEEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] := SRC1[127:0]
  1: TMP_DEST[127:0] := SRC1[255:128]
ESAC.
```

FI;

IF VL = 512

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

FI;

FOR j := 0 TO 3

i := j \* 32

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

THEN DEST[i+31:i] := TMP\_DEST[i+31:i]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:128] := 0

#### **VEEXTRACTI32x4 (EVEEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

```
CASE (imm8[0]) OF
  0: TMP_DEST[127:0] := SRC1[127:0]
  1: TMP_DEST[127:0] := SRC1[255:128]
ESAC.
```

FI;

IF VL = 512

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

```

FI;

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

```

**VEXTRACTI64x2 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

```

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

```

```

FI;
IF VL = 512

```

```

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

```

```

FI;

FOR j := 0 TO 1

```

```

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

```

```

  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]
  1: TMP_DEST[127:0] := SRC1[255:128]
ESAC.

```

```

FI;
IF VL = 512

```

```

CASE (imm8[1:0]) OF
  00: TMP_DEST[127:0] := SRC1[127:0]
  01: TMP_DEST[127:0] := SRC1[255:128]
  10: TMP_DEST[127:0] := SRC1[383:256]

```



```

    11: TMP_DEST[127:0] := SRC1[511:384]
  ESAC.
FI;

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

```

#### **VEEXTRACTI32x8 (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

```

#### **VEEXTRACTI32x8 (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

```

#### **VEEXTRACTI64x4 (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

```

**VEEXTRACTI64x4 (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

```

**VEEXTRACTI128 (memory destination form)**

```

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

```

**VEEXTRACTI128 (register destination form)**

```

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

```

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

```

VEEXTRACTI32x4 __m128i_mm512_extracti32x4_epi32(__m512i a, const int nid);
VEEXTRACTI32x4 __m128i_mm512_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m512i a, const int nid);
VEEXTRACTI32x4 __m128i_mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nid);
VEEXTRACTI32x4 __m128i_mm256_extracti32x4_epi32(__m256i a, const int nid);
VEEXTRACTI32x4 __m128i_mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nid);
VEEXTRACTI32x4 __m128i_mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nid);
VEEXTRACTI32x8 __m256i_mm512_extracti32x8_epi32(__m512i a, const int nid);
VEEXTRACTI32x8 __m256i_mm512_mask_extracti32x8_epi32(__m256i s, __mmask8 k, __m512i a, const int nid);
VEEXTRACTI32x8 __m256i_mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, const int nid);
VEEXTRACTI64x2 __m128i_mm512_extracti64x2_epi64(__m512i a, const int nid);
VEEXTRACTI64x2 __m128i_mm512_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m512i a, const int nid);
VEEXTRACTI64x2 __m128i_mm512_maskz_extracti64x2_epi64(__mmask8 k, __m512i a, const int nid);
VEEXTRACTI64x2 __m128i_mm256_extracti64x2_epi64(__m256i a, const int nid);

```

VEXTRACTI64x2 \_\_m128i \_\_mm256\_mask\_extracti64x2\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m256i a, const int nidx);  
 VEXTRACTI64x2 \_\_m128i \_\_mm256\_maskz\_extracti64x2\_epi64(\_\_mmask8 k, \_\_m256i a, const int nidx);  
 VEXTRACTI64x4 \_\_m256i \_\_mm512\_extracti64x4\_epi64(\_\_m512i a, const int nidx);  
 VEXTRACTI64x4 \_\_m256i \_\_mm512\_mask\_extracti64x4\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI64x4 \_\_m256i \_\_mm512\_maskz\_extracti64x4\_epi64(\_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI128 \_\_m128i \_\_mm256\_extracti128\_si256(\_\_m256i a, int offset);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

VEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions.”

EVEX-encoded instructions, see Table 2-54, “Type E6NF Class Exception Conditions.”

Additionally:

#UD IF VEX.L = 0.

#UD If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

## VFCMADDCPH/VFMADDCPH—Complex Multiply and Accumulate FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|-----------|------------------------------|-------------------------|--|
| EVEX.128.F2.MAP6.W0 56 /r<br>VFCMADDCPH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Complex multiply a pair of FP16 values from xmm2 and complex conjugate of xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1. |
| EVEX.256.F2.MAP6.W0 56 /r<br>VFCMADDCPH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Complex multiply a pair of FP16 values from ymm2 and complex conjugate of ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1. |
| EVEX.512.F2.MAP6.W0 56 /r<br>VFCMADDCPH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst {er} | A         | V/V                          | AVX512-FP16             | Complex multiply a pair of FP16 values from zmm2 and complex conjugate of zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1. |
| EVEX.128.F3.MAP6.W0 56 /r<br>VFMADDCPH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and store the result in xmm1 subject to writemask k1.                      |
| EVEX.256.F3.MAP6.W0 56 /r<br>VFMADDCPH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and store the result in ymm1 subject to writemask k1.                      |
| EVEX.512.F3.MAP6.W0 56 /r<br>VFMADDCPH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst {er}  | A         | V/V                          | AVX512-FP16             | Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and store the result in zmm1 subject to writemask k1.                      |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1        | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|------------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (r, w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

**Operation****VFCMADDCPH dest{k1}, src1, src2 (AVX512)**

VL = 128, 256, 512

KL := VL / 32

FOR i := 0 to KL-1:

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

IF broadcasting and src2 is memory:

tsrc2.fp16[2\*i+0] := src2.fp16[0]

tsrc2.fp16[2\*i+1] := src2.fp16[1]

ELSE:

tsrc2.fp16[2\*i+0] := src2.fp16[2\*i+0]

tsrc2.fp16[2\*i+1] := src2.fp16[2\*i+1]

FOR i := 0 to KL-1:

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

tmp[2\*i+0] := dest.fp16[2\*i+0] + src1.fp16[2\*i+0] \* tsrc2.fp16[2\*i+0]

tmp[2\*i+1] := dest.fp16[2\*i+1] + src1.fp16[2\*i+1] \* tsrc2.fp16[2\*i+0]

FOR i := 0 to KL-1:

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

// conjugate version subtracts odd final term

dest.fp16[2\*i+0] := tmp[2\*i+0] + src1.fp16[2\*i+1] \* tsrc2.fp16[2\*i+1]

dest.fp16[2\*i+1] := tmp[2\*i+1] - src1.fp16[2\*i+0] \* tsrc2.fp16[2\*i+1]

ELSE IF \*zeroing\*:

dest.fp16[2\*i+0] := 0

dest.fp16[2\*i+1] := 0

DEST[MAXVL-1:VL] := 0

**VMADDCPH dest{k1}, src1, src2 (AVX512)**

VL = 128, 256, 512

KL := VL / 32

FOR i := 0 to KL-1:

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

IF broadcasting and src2 is memory:

tsrc2.fp16[2\*i+0] := src2.fp16[0]

tsrc2.fp16[2\*i+1] := src2.fp16[1]

ELSE:

tsrc2.fp16[2\*i+0] := src2.fp16[2\*i+0]

tsrc2.fp16[2\*i+1] := src2.fp16[2\*i+1]

FOR i := 0 to KL-1:

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

tmp[2\*i+0] := dest.fp16[2\*i+0] + src1.fp16[2\*i+0] \* tsrc2.fp16[2\*i+0]

tmp[2\*i+1] := dest.fp16[2\*i+1] + src1.fp16[2\*i+1] \* tsrc2.fp16[2\*i+0]

FOR i := 0 to KL-1:

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

// non-conjugate version subtracts even term

dest.fp16[2\*i+0] := tmp[2\*i+0] - src1.fp16[2\*i+1] \* tsrc2.fp16[2\*i+1]

dest.fp16[2\*i+1] := tmp[2\*i+1] + src1.fp16[2\*i+0] \* tsrc2.fp16[2\*i+1]

ELSE IF \*zeroing\*:

```
dest.fp16[2*i+0] := 0
dest.fp16[2*i+1] := 0
```

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

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

```
VFCMADDCPH __m128h __mm_fcmadd_pch (__m128h a, __m128h b, __m128h c);
VFCMADDCPH __m128h __mm_mask_fcmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFCMADDCPH __m128h __mm_mask3_fcmadd_pch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFCMADDCPH __m128h __mm_maskz_fcmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFCMADDCPH __m256h __mm256_fcmadd_pch (__m256h a, __m256h b, __m256h c);
VFCMADDCPH __m256h __mm256_mask_fcmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFCMADDCPH __m256h __mm256_mask3_fcmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFCMADDCPH __m256h __mm256_maskz_fcmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFCMADDCPH __m512h __mm512_fcmadd_pch (__m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h __mm512_mask_fcmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFCMADDCPH __m512h __mm512_mask3_fcmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFCMADDCPH __m512h __mm512_maskz_fcmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFCMADDCPH __m512h __mm512_fcmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h __mm512_mask_fcmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFCMADDCPH __m512h __mm512_mask3_fcmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFCMADDCPH __m512h __mm512_maskz_fcmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);

VFMADDCPH __m128h __mm_fmadd_pch (__m128h a, __m128h b, __m128h c);
VFMADDCPH __m128h __mm_mask_fmadd_pch (__m128h a, __mmask8 k, __m128h b, __m128h c);
VFMADDCPH __m128h __mm_mask3_fmadd_pch (__m128h a, __m128h b, __m128h c, __mmask8 k);
VFMADDCPH __m128h __mm_maskz_fmadd_pch (__mmask8 k, __m128h a, __m128h b, __m128h c);
VFMADDCPH __m256h __mm256_fmadd_pch (__m256h a, __m256h b, __m256h c);
VFMADDCPH __m256h __mm256_mask_fmadd_pch (__m256h a, __mmask8 k, __m256h b, __m256h c);
VFMADDCPH __m256h __mm256_mask3_fmadd_pch (__m256h a, __m256h b, __m256h c, __mmask8 k);
VFMADDCPH __m256h __mm256_maskz_fmadd_pch (__mmask8 k, __m256h a, __m256h b, __m256h c);
VFMADDCPH __m512h __mm512_fmadd_pch (__m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h __mm512_mask_fmadd_pch (__m512h a, __mmask16 k, __m512h b, __m512h c);
VFMADDCPH __m512h __mm512_mask3_fmadd_pch (__m512h a, __m512h b, __m512h c, __mmask16 k);
VFMADDCPH __m512h __mm512_maskz_fmadd_pch (__mmask16 k, __m512h a, __m512h b, __m512h c);
VFMADDCPH __m512h __mm512_fmadd_round_pch (__m512h a, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h __mm512_mask_fmadd_round_pch (__m512h a, __mmask16 k, __m512h b, __m512h c, const int rounding);
VFMADDCPH __m512h __mm512_mask3_fmadd_round_pch (__m512h a, __m512h b, __m512h c, __mmask16 k, const int rounding);
VFMADDCPH __m512h __mm512_maskz_fmadd_round_pch (__mmask16 k, __m512h a, __m512h b, __m512h c, const int rounding);
```

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

### Other Exceptions

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

Additionally:

#UD If (dest\_reg == src1\_reg) or (dest\_reg == src2\_reg).

## VFCMADDCSH/VFMADDCSH—Complex Multiply and Accumulate Scalar FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F2.MAP6.W0 57 /r<br>VFCMADDCSH xmm1{k1}{z}, xmm2,<br>xmm3/m32 {er} | A         | V/V                          | AVX512-FP16           | Complex multiply a pair of FP16 values from xmm2 and complex conjugate of xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32]. |
| EVEX.LLIG.F3.MAP6.W0 57 /r<br>VFMADDCSH xmm1{k1}{z}, xmm2,<br>xmm3/m32 {er}  | A         | V/V                          | AVX512-FP16           | Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, add to xmm1 and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].                      |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1        | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|------------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (r, w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a complex multiply and accumulate operation. There are normal and complex conjugate forms of the operation.

The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

### Operation

#### VFCMADDCSH dest{k1}, src1, src2 (AVX512)

IF k1[0] or \*no writemask\*:

```
tmp[0] := dest.fp16[0] + src1.fp16[0] * src2.fp16[0]
```

```
tmp[1] := dest.fp16[1] + src1.fp16[1] * src2.fp16[0]
```

```
// conjugate version subtracts odd final term
```

```
dest.fp16[0] := tmp[0] + src1.fp16[1] * src2.fp16[1]
```

```
dest.fp16[1] := tmp[1] - src1.fp16[0] * src2.fp16[1]
```

ELSE IF \*zeroing\*:

```
dest.fp16[0] := 0
```

```
dest.fp16[1] := 0
```

```
DEST[127:32] := src1[127:32] // copy upper part of src1
```

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

**VFMADDCSH dest{k1}, src1, src2 (AVX512)**

IF k1[0] or \*no writemask\*:

```
tmp[0] := dest.fp16[0] + src1.fp16[0] * src2.fp16[0]
tmp[1] := dest.fp16[1] + src1.fp16[1] * src2.fp16[0]
```

// non-conjugate version subtracts last even term

```
dest.fp16[0] := tmp[0] - src1.fp16[1] * src2.fp16[1]
```

```
dest.fp16[1] := tmp[1] + src1.fp16[0] * src2.fp16[1]
```

ELSE IF \*zeroing\*:

```
dest.fp16[0] := 0
```

```
dest.fp16[1] := 0
```

DEST[127:32] := src1[127:32] // copy upper part of src1

DEST[MAXVL-1:128] := 0

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

VFCMADDCSH \_\_m128h \_\_mm\_fcmadd\_round\_sch (\_\_m128h a, \_\_m128h b, \_\_m128h c, const int rounding);

VFCMADDCSH \_\_m128h \_\_mm\_mask\_fcmadd\_round\_sch (\_\_m128h a, \_\_mmask8 k, \_\_m128h b, \_\_m128h c, const int rounding);

VFCMADDCSH \_\_m128h \_\_mm\_mask3\_fcmadd\_round\_sch (\_\_m128h a, \_\_m128h b, \_\_m128h c, \_\_mmask8 k, const int rounding);

VFCMADDCSH \_\_m128h \_\_mm\_maskz\_fcmadd\_round\_sch (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, \_\_m128h c, const int rounding);

VFCMADDCSH \_\_m128h \_\_mm\_fcmadd\_sch (\_\_m128h a, \_\_m128h b, \_\_m128h c);

VFCMADDCSH \_\_m128h \_\_mm\_mask\_fcmadd\_sch (\_\_m128h a, \_\_mmask8 k, \_\_m128h b, \_\_m128h c);

VFCMADDCSH \_\_m128h \_\_mm\_mask3\_fcmadd\_sch (\_\_m128h a, \_\_m128h b, \_\_m128h c, \_\_mmask8 k);

VFCMADDCSH \_\_m128h \_\_mm\_maskz\_fcmadd\_sch (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, \_\_m128h c);

VFMADDCSH \_\_m128h \_\_mm\_fmadd\_round\_sch (\_\_m128h a, \_\_m128h b, \_\_m128h c, const int rounding);

VFMADDCSH \_\_m128h \_\_mm\_mask\_fmadd\_round\_sch (\_\_m128h a, \_\_mmask8 k, \_\_m128h b, \_\_m128h c, const int rounding);

VFMADDCSH \_\_m128h \_\_mm\_mask3\_fmadd\_round\_sch (\_\_m128h a, \_\_m128h b, \_\_m128h c, \_\_mmask8 k, const int rounding);

VFMADDCSH \_\_m128h \_\_mm\_maskz\_fmadd\_round\_sch (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, \_\_m128h c, const int rounding);

VFMADDCSH \_\_m128h \_\_mm\_fmadd\_sch (\_\_m128h a, \_\_m128h b, \_\_m128h c);

VFMADDCSH \_\_m128h \_\_mm\_mask\_fmadd\_sch (\_\_m128h a, \_\_mmask8 k, \_\_m128h b, \_\_m128h c);

VFMADDCSH \_\_m128h \_\_mm\_mask3\_fmadd\_sch (\_\_m128h a, \_\_m128h b, \_\_m128h c, \_\_mmask8 k);

VFMADDCSH \_\_m128h \_\_mm\_maskz\_fmadd\_sch (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, \_\_m128h c);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-58, "Type E10 Class Exception Conditions."

Additionally:

#UD If (dest\_reg == src1\_reg) or (dest\_reg == src2\_reg).



## VFCMULCPH/VFMULCPH—Complex Multiply FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.F2.MAP6.W0 D6 /r<br>VFCMULCPH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Complex multiply a pair of FP16 values from xmm2 and complex conjugate of xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1. |
| EVEX.256.F2.MAP6.W0 D6 /r<br>VFCMULCPH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Complex multiply a pair of FP16 values from ymm2 and complex conjugate of ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1. |
| EVEX.512.F2.MAP6.W0 D6 /r<br>VFCMULCPH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst {er} | A         | V/V                          | AVX512-FP16             | Complex multiply a pair of FP16 values from zmm2 and complex conjugate of zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1. |
| EVEX.128.F3.MAP6.W0 D6 /r<br>VFMULCPH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Complex multiply a pair of FP16 values from xmm2 and xmm3/m128/m32bcst, and store the result in xmm1 subject to writemask k1.                      |
| EVEX.256.F3.MAP6.W0 D6 /r<br>VFMULCPH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Complex multiply a pair of FP16 values from ymm2 and ymm3/m256/m32bcst, and store the result in ymm1 subject to writemask k1.                      |
| EVEX.512.F3.MAP6.W0 D6 /r<br>VFMULCPH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst {er}  | A         | V/V                          | AVX512-FP16             | Complex multiply a pair of FP16 values from zmm2 and zmm3/m512/m32bcst, and store the result in zmm1 subject to writemask k1.                      |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|---------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The broadcasting and masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

### Operation

**VFCMULCPH dest{k1}, src1, src2 (AVX512)**

VL = 128, 256 or 512

KL := VL/32

FOR i := 0 to KL-1:

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

IF broadcasting and src2 is memory:

tsrc2.fp16[2\*i+0] := src2.fp16[0]

tsrc2.fp16[2\*i+1] := src2.fp16[1]

ELSE:

tsrc2.fp16[2\*i+0] := src2.fp16[2\*i+0]

tsrc2.fp16[2\*i+1] := src2.fp16[2\*i+1]

FOR i := 0 to KL-1:

```

IF k1[j] or *no writemask*:
    tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
    tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

```

```

FOR i := 0 to KL-1:

```

```

    IF k1[j] or *no writemask*:
        // conjugate version subtracts odd final term
        dest.fp16[2*i] := tmp.fp16[2*i+0] + src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp.fp16[2*i+1] - src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

```

```

DEST[MAXVL-1:VL] := 0

```

### VFMULCPH dest{k1}, src1, src2 (AVX512)

VL = 128, 256 or 512

KL := VL/32

```

FOR i := 0 to KL-1:

```

```

    IF k1[j] or *no writemask*:
        IF broadcasting and src2 is memory:
            tsrc2.fp16[2*i+0] := src2.fp16[0]
            tsrc2.fp16[2*i+1] := src2.fp16[1]
        ELSE:
            tsrc2.fp16[2*i+0] := src2.fp16[2*i+0]
            tsrc2.fp16[2*i+1] := src2.fp16[2*i+1]

```

```

FOR i := 0 to kl-1:

```

```

    IF k1[j] or *no writemask*:
        tmp.fp16[2*i+0] := src1.fp16[2*i+0] * tsrc2.fp16[2*i+0]
        tmp.fp16[2*i+1] := src1.fp16[2*i+1] * tsrc2.fp16[2*i+0]

```

```

FOR i := 0 to KL-1:

```

```

    IF k1[j] or *no writemask*:
        // non-conjugate version subtracts last even term
        dest.fp16[2*i+0] := tmp.fp16[2*i+0] - src1.fp16[2*i+1] * tsrc2.fp16[2*i+1]
        dest.fp16[2*i+1] := tmp.fp16[2*i+1] + src1.fp16[2*i+0] * tsrc2.fp16[2*i+1]
    ELSE IF *zeroing*:
        dest.fp16[2*i+0] := 0
        dest.fp16[2*i+1] := 0

```

```

DEST[MAXVL-1:VL] := 0

```

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

```

VFCMULCPH __m128h __mm_cmul_pch (__m128h a, __m128h b);
VFCMULCPH __m128h __mm_mask_cmul_pch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m128h __mm_maskz_cmul_pch (__mmask8 k, __m128h a, __m128h b);
VFCMULCPH __m256h __mm256_cmul_pch (__m256h a, __m256h b);
VFCMULCPH __m256h __mm256_mask_cmul_pch (__m256h src, __mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m256h __mm256_maskz_cmul_pch (__mmask8 k, __m256h a, __m256h b);
VFCMULCPH __m512h __mm512_cmul_pch (__m512h a, __m512h b);
VFCMULCPH __m512h __mm512_mask_cmul_pch (__m512h src, __mmask16 k, __m512h a, __m512h b);
VFCMULCPH __m512h __mm512_maskz_cmul_pch (__mmask16 k, __m512h a, __m512h b);

```

VFCMULCPH \_\_m512h \_\_mm512\_cmul\_round\_pch (\_\_m512h a, \_\_m512h b, const int rounding);  
 VFCMULCPH \_\_m512h \_\_mm512\_mask\_cmul\_round\_pch (\_\_m512h src, \_\_mmask16 k, \_\_m512h a, \_\_m512h b, const int rounding);  
 VFCMULCPH \_\_m512h \_\_mm512\_maskz\_cmul\_round\_pch (\_\_mmask16 k, \_\_m512h a, \_\_m512h b, const int rounding);  
 VFCMULCPH \_\_m128h \_\_mm\_fcmul\_pch (\_\_m128h a, \_\_m128h b);  
 VFCMULCPH \_\_m128h \_\_mm\_mask\_fcmul\_pch (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
 VFCMULCPH \_\_m128h \_\_mm\_maskz\_fcmul\_pch (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
 VFCMULCPH \_\_m256h \_\_mm256\_fcmul\_pch (\_\_m256h a, \_\_m256h b);  
 VFCMULCPH \_\_m256h \_\_mm256\_mask\_fcmul\_pch (\_\_m256h src, \_\_mmask8 k, \_\_m256h a, \_\_m256h b);  
 VFCMULCPH \_\_m256h \_\_mm256\_maskz\_fcmul\_pch (\_\_mmask8 k, \_\_m256h a, \_\_m256h b);  
 VFCMULCPH \_\_m512h \_\_mm512\_fcmul\_pch (\_\_m512h a, \_\_m512h b);  
 VFCMULCPH \_\_m512h \_\_mm512\_mask\_fcmul\_pch (\_\_m512h src, \_\_mmask16 k, \_\_m512h a, \_\_m512h b);  
 VFCMULCPH \_\_m512h \_\_mm512\_maskz\_fcmul\_pch (\_\_mmask16 k, \_\_m512h a, \_\_m512h b);  
 VFCMULCPH \_\_m512h \_\_mm512\_fcmul\_round\_pch (\_\_m512h a, \_\_m512h b, const int rounding);  
 VFCMULCPH \_\_m512h \_\_mm512\_mask\_fcmul\_round\_pch (\_\_m512h src, \_\_mmask16 k, \_\_m512h a, \_\_m512h b, const int rounding);  
 VFCMULCPH \_\_m512h \_\_mm512\_maskz\_fcmul\_round\_pch (\_\_mmask16 k, \_\_m512h a, \_\_m512h b, const int rounding);

VFMULCPH \_\_m128h \_\_mm\_fmuls\_pch (\_\_m128h a, \_\_m128h b);  
 VFMULCPH \_\_m128h \_\_mm\_mask\_fmuls\_pch (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
 VFMULCPH \_\_m128h \_\_mm\_maskz\_fmuls\_pch (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
 VFMULCPH \_\_m256h \_\_mm256\_fmuls\_pch (\_\_m256h a, \_\_m256h b);  
 VFMULCPH \_\_m256h \_\_mm256\_mask\_fmuls\_pch (\_\_m256h src, \_\_mmask8 k, \_\_m256h a, \_\_m256h b);  
 VFMULCPH \_\_m256h \_\_mm256\_maskz\_fmuls\_pch (\_\_mmask8 k, \_\_m256h a, \_\_m256h b);  
 VFMULCPH \_\_m512h \_\_mm512\_fmuls\_pch (\_\_m512h a, \_\_m512h b);  
 VFMULCPH \_\_m512h \_\_mm512\_mask\_fmuls\_pch (\_\_m512h src, \_\_mmask16 k, \_\_m512h a, \_\_m512h b);  
 VFMULCPH \_\_m512h \_\_mm512\_maskz\_fmuls\_pch (\_\_mmask16 k, \_\_m512h a, \_\_m512h b);  
 VFMULCPH \_\_m512h \_\_mm512\_fmuls\_round\_pch (\_\_m512h a, \_\_m512h b, const int rounding);  
 VFMULCPH \_\_m512h \_\_mm512\_mask\_fmuls\_round\_pch (\_\_m512h src, \_\_mmask16 k, \_\_m512h a, \_\_m512h b, const int rounding);  
 VFMULCPH \_\_m512h \_\_mm512\_maskz\_fmuls\_round\_pch (\_\_mmask16 k, \_\_m512h a, \_\_m512h b, const int rounding);  
 VFMULCPH \_\_m128h \_\_mm\_mask\_fmuls\_pch (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
 VFMULCPH \_\_m128h \_\_mm\_maskz\_fmuls\_pch (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
 VFMULCPH \_\_m128h \_\_mm\_fmuls\_pch (\_\_m128h a, \_\_m128h b);  
 VFMULCPH \_\_m256h \_\_mm256\_mask\_fmuls\_pch (\_\_m256h src, \_\_mmask8 k, \_\_m256h a, \_\_m256h b);  
 VFMULCPH \_\_m256h \_\_mm256\_maskz\_fmuls\_pch (\_\_mmask8 k, \_\_m256h a, \_\_m256h b);  
 VFMULCPH \_\_m256h \_\_mm256\_fmuls\_pch (\_\_m256h a, \_\_m256h b);  
 VFMULCPH \_\_m512h \_\_mm512\_mask\_fmuls\_pch (\_\_m512h src, \_\_mmask16 k, \_\_m512h a, \_\_m512h b);  
 VFMULCPH \_\_m512h \_\_mm512\_maskz\_fmuls\_pch (\_\_mmask16 k, \_\_m512h a, \_\_m512h b);  
 VFMULCPH \_\_m512h \_\_mm512\_fmuls\_pch (\_\_m512h a, \_\_m512h b);  
 VFMULCPH \_\_m512h \_\_mm512\_mask\_fmuls\_round\_pch (\_\_m512h src, \_\_mmask16 k, \_\_m512h a, \_\_m512h b, const int rounding);  
 VFMULCPH \_\_m512h \_\_mm512\_maskz\_fmuls\_round\_pch (\_\_mmask16 k, \_\_m512h a, \_\_m512h b, const int rounding);  
 VFMULCPH \_\_m512h \_\_mm512\_fmuls\_round\_pch (\_\_m512h a, \_\_m512h b, const int rounding);

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

### Other Exceptions

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

Additionally:

#UD                    If (dest\_reg == src1\_reg) or (dest\_reg == src2\_reg).

## VFCMULCSH/VFMULCSH—Complex Multiply Scalar FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|---|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F2.MAP6.W0 D7 /r<br>VFCMULCSH xmm1{k1}{z}, xmm2,<br>xmm3/m32 {er} | A         | V/V                          | AVX512-FP16           | Complex multiply a pair of FP16 values from xmm2 and complex conjugate of xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32]. |
| EVEX.LLIG.F3.MAP6.W0 D7 /r<br>VFMULCSH xmm1{k1}{z}, xmm2,<br>xmm3/m32 {er}  | A         | V/V                          | AVX512-FP16           | Complex multiply a pair of FP16 values from xmm2 and xmm3/m32, and store the result in xmm1 subject to writemask k1. Bits 127:32 of xmm2 are copied to xmm1[127:32].                      |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a complex multiply operation. There are normal and complex conjugate forms of the operation. The masking for this operation is done on 32-bit quantities representing a pair of FP16 values.

Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Rounding is performed at every FMA (fused multiply and add) boundary. Execution occurs as if all MXCSR exceptions are masked. MXCSR status bits are updated to reflect exceptional conditions.

### Operation

**VFCMULCSH dest{k1}, src1, src2 (AVX512)**

KL := VL / 32

IF k1[0] or \*no writemask\*:

```
tmp.fp16[0] := src1.fp16[0] * src2.fp16[0]
tmp.fp16[1] := src1.fp16[1] * src2.fp16[0]
```

// conjugate version subtracts odd final term

```
dest.fp16[0] := tmp.fp16[0] + src1.fp16[1] * src2.fp16[1]
dest.fp16[1] := tmp.fp16[1] - src1.fp16[0] * src2.fp16[1]
```

ELSE IF \*zeroing\*:

```
dest.fp16[0] := 0
dest.fp16[1] := 0
```

DEST[127:32] := src1[127:32] // copy upper part of src1

DEST[MAXVL-1:128] := 0

**VFMULCSH dest{k1}, src1, src2 (AVX512)**

KL := VL / 32

IF k1[0] or \*no writemask\*:

```
// non-conjugate version subtracts last even term
tmp.fp16[0] := src1.fp16[0] * src2.fp16[0]
tmp.fp16[1] := src1.fp16[1] * src2.fp16[0]
dest.fp16[0] := tmp.fp16[0] - src1.fp16[1] * src2.fp16[1]
dest.fp16[1] := tmp.fp16[1] + src1.fp16[0] * src2.fp16[1]
```

ELSE IF \*zeroing\*:

```
dest.fp16[0] := 0
dest.fp16[1] := 0
```

DEST[127:32] := src1[127:32] // copy upper part of src1

DEST[MAXVL-1:128] := 0

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

```
VFCMULCSH __m128h __mm_cmul_round_sch (__m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h __mm_mask_cmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h __mm_maskz_cmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h __mm_cmul_sch (__m128h a, __m128h b);
VFCMULCSH __m128h __mm_mask_cmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h __mm_maskz_cmul_sch (__mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h __mm_fcmul_round_sch (__m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h __mm_mask_fcmul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h __mm_maskz_fcmul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFCMULCSH __m128h __mm_fcmul_sch (__m128h a, __m128h b);
VFCMULCSH __m128h __mm_mask_fcmul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFCMULCSH __m128h __mm_maskz_fcmul_sch (__mmask8 k, __m128h a, __m128h b);
```

```
VFMULCSH __m128h __mm_fmuls_round_sch (__m128h a, __m128h b, const int rounding);
VFMULCSH __m128h __mm_mask_fmuls_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h __mm_maskz_fmuls_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h __mm_fmuls_sch (__m128h a, __m128h b);
VFMULCSH __m128h __mm_mask_fmuls_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h __mm_maskz_fmuls_sch (__mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h __mm_mask_mul_round_sch (__m128h src, __mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h __mm_maskz_mul_round_sch (__mmask8 k, __m128h a, __m128h b, const int rounding);
VFMULCSH __m128h __mm_mul_round_sch (__m128h a, __m128h b, const int rounding);
VFMULCSH __m128h __mm_mask_mul_sch (__m128h src, __mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h __mm_maskz_mul_sch (__mmask8 k, __m128h a, __m128h b);
VFMULCSH __m128h __mm_mul_sch (__m128h a, __m128h b);
```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-58, “Type E10 Class Exception Conditions.”

Additionally:

#UD If (dest\_reg == src1\_reg) or (dest\_reg == src2\_reg).

## VFIXUPIMMPD—Fix Up Special Packed Float64 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W1 54 /r ib<br>VFIXUPIMMPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8      | A         | V/V                          | AVX512VL<br>AVX512F      | 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.              |
| EVEX.256.66.0F3A.W1 54 /r ib<br>VFIXUPIMMPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8      | A         | V/V                          | AVX512VL<br>AVX512F      | 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.              |
| EVEX.512.66.0F3A.W1 54 /r ib<br>VFIXUPIMMPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{sae}, imm8 | A         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

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

### 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 double precision floating-point input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

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

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

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

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

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

**Operation**

```
enum TOKEN_TYPE
```

```
{
    QNAN_TOKEN := 0,
    SNAN_TOKEN := 1,
    ZERO_VALUE_TOKEN := 2,
    POS_ONE_VALUE_TOKEN := 3,
    NEG_INF_TOKEN := 4,
    POS_INF_TOKEN := 5,
    NEG_VALUE_TOKEN := 6,
    POS_VALUE_TOKEN := 7
}
```

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

; The required response from src3 table is extracted

```
token_response[3:0] = tbl3[3+4*j:4*j];
```

```
CASE(token_response[3:0]) {
    0000: dest[63:0] := dest[63:0]; ; preserve content of DEST
    0001: dest[63:0] := tsrc[63:0]; ; pass through src1 normal input value, denormal as zero
    0010: dest[63:0] := QNaN(tsrc[63:0]);
    0011: dest[63:0] := QNaN_Indefinite;
    0100: dest[63:0] := -INF;
    0101: dest[63:0] := +INF;
    0110: dest[63:0] := tsrc.sign? -INF : +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[j+63:i], SRC2[63:0], imm8 [7:0])

        ELSE

          DEST[i+63:i] := FIXUPIMM\_DP(DEST[i+63:i], SRC1[j+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

  FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

Immediate Control Description:

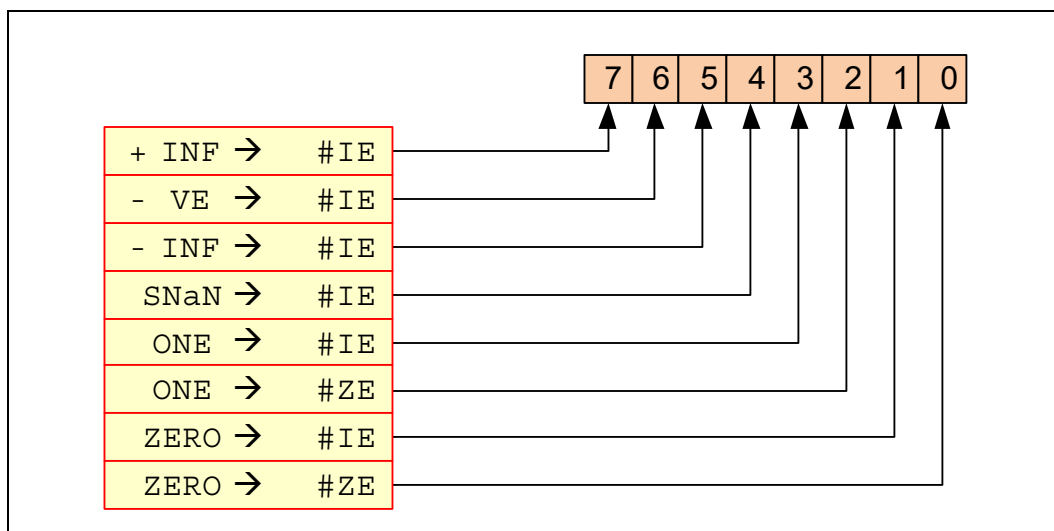


Figure 5-9. VFIXUPIMMPD Immediate Control Description



**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 imm, int sae);  
 VFIXUPIMMPD \_\_m512d \_\_mm512\_mask\_fixupimm\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, \_\_m512i tbl, int imm, int sae);  
 VFIXUPIMMPD \_\_m512d \_\_mm512\_maskz\_fixupimm\_round\_pd( \_\_mmask8 k, \_\_m512d a, \_\_m512i tbl, int imm, int sae);  
 VFIXUPIMMPD \_\_m256d \_\_mm256\_fixupimm\_pd( \_\_m256d a, m256d b, \_\_m256i c, int imm8);  
 VFIXUPIMMPD \_\_m256d \_\_mm256\_mask\_fixupimm\_pd(\_\_m256d a, \_\_mmask8 k, \_\_m256d b, \_\_m256i c, int imm8);  
 VFIXUPIMMPD \_\_m256d \_\_mm256\_maskz\_fixupimm\_pd( \_\_mmask8 k, \_\_m256d a, \_\_m256d b, \_\_m256i c, int imm8);  
 VFIXUPIMMPD \_\_m128d \_\_mm\_fixupimm\_pd( \_\_m128d a, \_\_m128d b, \_\_m128i c, int imm8);  
 VFIXUPIMMPD \_\_m128d \_\_mm\_mask\_fixupimm\_pd(\_\_m128d a, \_\_mmask8 k, \_\_m128d b, \_\_m128i c, int imm8);  
 VFIXUPIMMPD \_\_m128d \_\_mm\_maskz\_fixupimm\_pd( \_\_mmask8 k, \_\_m128d a, \_\_m128d b, 128ic, int imm8);

**SIMD Floating-Point Exceptions**

Zero, Invalid.

**Other Exceptions**

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

## VFIXUPIMMPS—Fix Up Special Packed Float32 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W0 54 /r<br>VFIXUPIMMPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512F      | 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.              |
| EVEX.256.66.0F3A.W0 54 /r<br>VFIXUPIMMPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512F      | 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.              |
| EVEX.512.66.0F3A.W0 54 /r ib<br>VFIXUPIMMPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{sae}, imm8 | A         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

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

### 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 single precision floating-point 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, 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], tbi3[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] = tbi3[3+4*j:4*j];

CASE(token_response[3:0]) {
    0000: dest[31:0] := dest[31:0]; ; preserve content of DEST
    0001: dest[31:0] := tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
    0010: dest[31:0] := QNaN(tsrc[31:0]);
    0011: dest[31:0] := QNaN_Indefinite;
    0100: dest[31:0] := -INF;
    0101: dest[31:0] := +INF;
    0110: dest[31:0] := tsrc.sign? -INF : +INF;
    0111: dest[31:0] := -0;
    1000: dest[31:0] := +0;
    1001: dest[31:0] := -1;
    1010: dest[31:0] := +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:

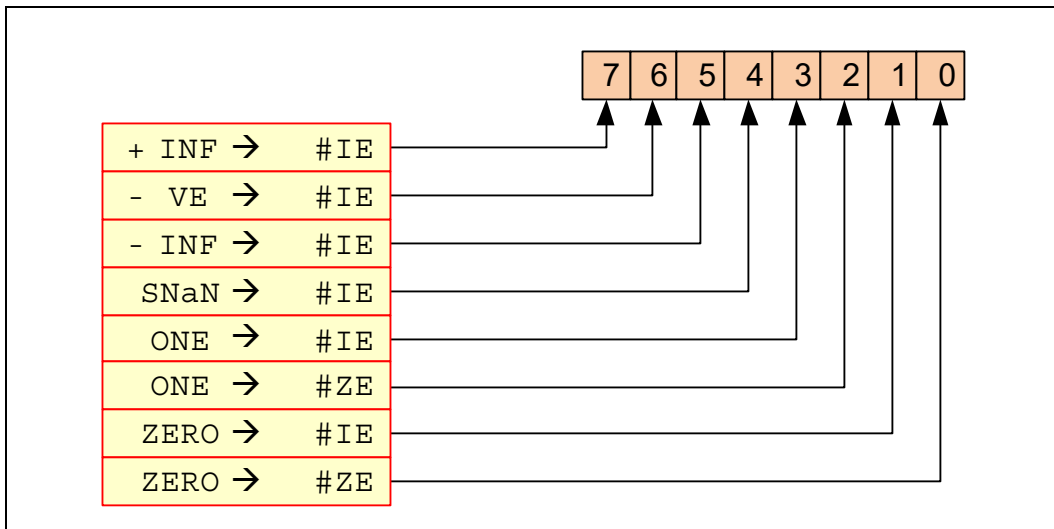


Figure 5-10. VFIXUPIMMPS Immediate Control Description

### 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, __m256b, __m256i c, int imm8);
VFIXUPIMMPS __m128 __mm_fixupimm_ps( __m128 a, __m128 b, __m128i c, int imm8);
VFIXUPIMMPS __m128 __mm_mask_fixupimm_ps(__m128 a, __mmask8 k, __m128 b, __m128i c, int imm8);
VFIXUPIMMPS __m128 __mm_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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.66.0F3A.W1 55 /r ib<br>VFIXUPIMMSD xmm1 {k1}{z},<br>xmm2, xmm3/m64{sae}, imm8 | A         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### 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 double precision floating-point 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, `VFIXUPIMMSD` 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_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;
  0110: dest[63:0] := tsrc.sign? -INF : +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
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

Immediate Control Description:

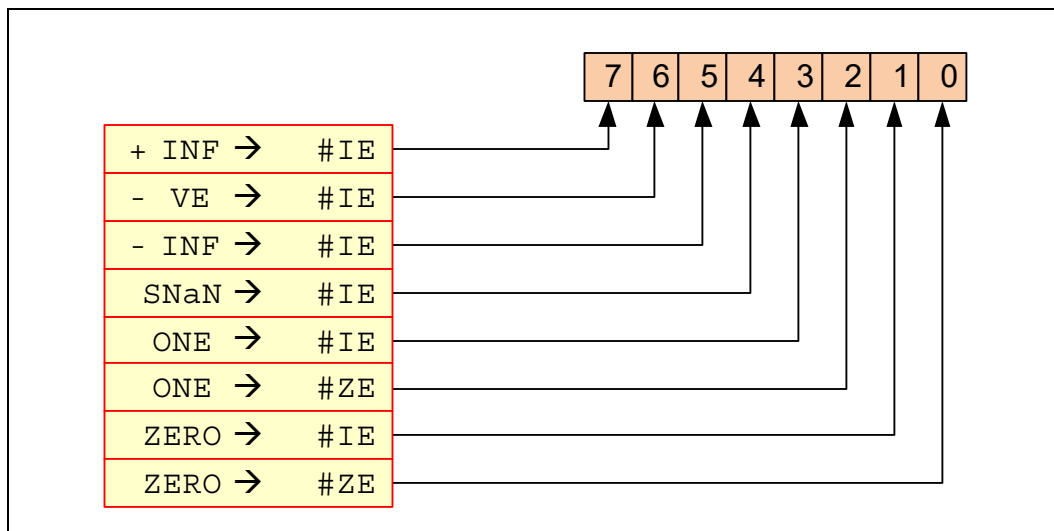


Figure 5-11. VFIXUPIMMSD Immediate Control Description

**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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.66.0F3A.W0 55 /r ib<br>VFIXUPIMMSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae}, imm8 | A         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### 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 single precision floating-point 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, `VFIXUPIMMSS` can be used after the N-R reciprocal sequence to set the result to the correct value (i.e., INF when the input is 0).

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

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

`MXCSR.DAZ` is used and refer to `zmm2` only (i.e., `zmm1` is not considered as zero in case `MXCSR.DAZ` is set).

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

### Operation

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

```

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

```

; The required response from src3 table is extracted

```
token_response[3:0] = tbl3[3+4*j:4*j];
```

```

CASE(token_response[3:0]) {
  0000: dest[31:0] := dest[31:0]; ; preserve content of DEST
  0001: dest[31:0] := tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[31:0] := QNaN(tsrc[31:0]);
  0011: dest[31:0] := QNaN_Indefinite;
  0100: dest[31:0] := -INF;
  0101: dest[31:0] := +INF;
  0110: dest[31:0] := tsrc.sign? -INF : +INF;
  0111: dest[31:0] := -0;
  1000: dest[31:0] := +0;
  1001: dest[31:0] := -1;
  1010: dest[31:0] := +1;
  1011: dest[31:0] := 1/2;
  1100: dest[31:0] := 90.0;
  1101: dest[31:0] := PI/2;
  1110: dest[31:0] := MAX_FLOAT;
  1111: dest[31:0] := -MAX_FLOAT;
} ; end of token_response CASE

```

; The required fault reporting from imm8 is extracted

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

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

```
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
```

```
 ; end fault reporting
```

```
return dest[31:0];
```

```
} ; end of FIXUPIMM_SP()
```

**VFIXUPIMMSS (EVEX encoded version)**

```

IF 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* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
      ELSE DEST[31:0] := 0 ; zeroing-masking
    FI
  FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

Immediate Control Description:

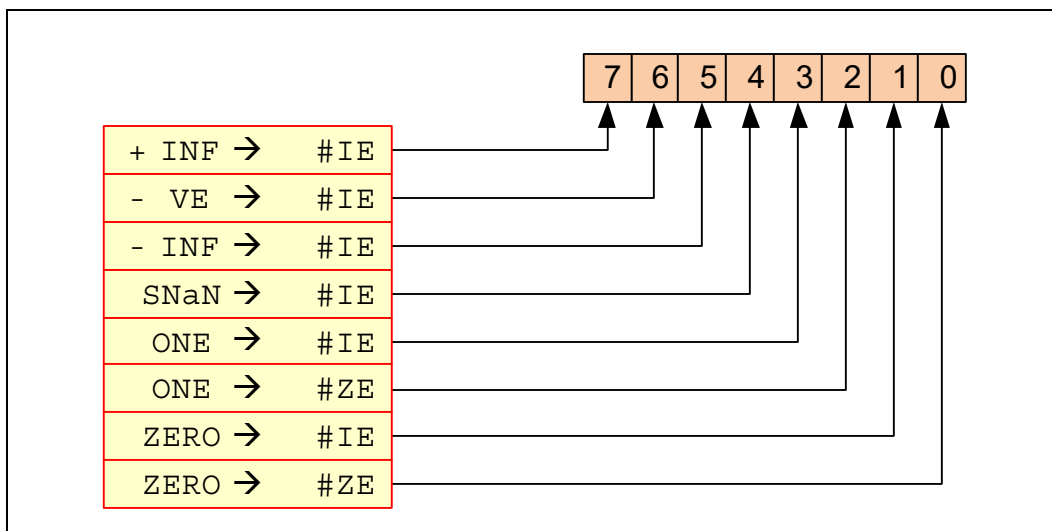


Figure 5-12. VFIXUPIMMSS Immediate Control Description

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

```

VFIXUPIMMSS __m128 __mm_fixupimm_ss( __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_ss( __m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_ss( __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 __mm_fixupimm_round_ss( __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_mask_fixupimm_round_ss( __m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 __mm_maskz_fixupimm_round_ss( __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Zero, Invalid

**Other Exceptions**

See Table 2-47, "Type E3 Class Exception Conditions."

## VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W1 98 /r<br>VFMADD132PD xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.          |
| VEX.128.66.0F38.W1 A8 /r<br>VFMADD213PD xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.          |
| VEX.128.66.0F38.W1 B8 /r<br>VFMADD231PD xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.          |
| VEX.256.66.0F38.W1 98 /r<br>VFMADD132PD ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.          |
| VEX.256.66.0F38.W1 A8 /r<br>VFMADD213PD ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.          |
| VEX.256.66.0F38.W1 B8 /r<br>VFMADD231PD ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.          |
| EVEX.128.66.0F38.W1 98 /r<br>VFMADD132PD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1. |
| EVEX.128.66.0F38.W1 A8 /r<br>VFMADD213PD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1. |
| EVEX.128.66.0F38.W1 B8 /r<br>VFMADD231PD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1. |
| EVEX.256.66.0F38.W1 98 /r<br>VFMADD132PD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1. |
| EVEX.256.66.0F38.W1 A8 /r<br>VFMADD213PD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1. |
| EVEX.256.66.0F38.W1 B8 /r<br>VFMADD231PD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1. |
| EVEX.512.66.0F38.W1 98 /r<br>VFMADD132PD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1. |
| EVEX.512.66.0F38.W1 A8 /r<br>VFMADD213PD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed double precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1. |
| EVEX.512.66.0F38.W1 B8 /r<br>VFMADD231PD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1. |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## 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
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

```

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

```

IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    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

```

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

```

IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    DEST[n+63:n] := RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] := 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] := 0
FI

```

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 64

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

THEN DEST[i+63:i] :=

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] + SRC2[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

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

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

FOR j := 0 TO KL-1

i := j \* 64

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

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] :=

RoundFPControl\_MXCSR(DEST[i+63:i]\*SRC3[63:0] + SRC2[i+63:i])

ELSE

DEST[i+63:i] :=

RoundFPControl\_MXCSR(DEST[i+63:i]\*SRC3[i+63:i] + SRC2[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VFMAADD213PD 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

**VFMAADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j := 0 TO KL-1

i := j \* 64

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

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] :=

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[63:0])

ELSE

DEST[i+63:i] :=

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0



**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

IF (VL = 512) AND (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[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**

```

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 __m512d __mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, 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 __mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d __mm256_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.”

## VF[,N]MADD[132,213,231]PH—Fused Multiply-Add of Packed FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP6.W0 98 /r<br>VFMADD132PH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm3/m128/m16bcst, add to xmm2, and store<br>the result in xmm1.                                     |
| EVEX.256.66.MAP6.W0 98 /r<br>VFMADD132PH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm3/m256/m16bcst, add to ymm2, and store<br>the result in ymm1.                                     |
| EVEX.512.66.MAP6.W0 98 /r<br>VFMADD132PH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m16bcst {er}  | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and<br>zmm3/m512/m16bcst, add to zmm2, and store<br>the result in zmm1.                                     |
| EVEX.128.66.MAP6.W0 A8 /r<br>VFMADD213PH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm2, add to xmm3/m128/m16bcst, and store<br>the result in xmm1.                                     |
| EVEX.256.66.MAP6.W0 A8 /r<br>VFMADD213PH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm2, add to ymm3/m256/m16bcst, and store<br>the result in ymm1.                                     |
| EVEX.512.66.MAP6.W0 A8 /r<br>VFMADD213PH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m16bcst {er}  | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and<br>zmm2, add to zmm3/m512/m16bcst, and store<br>the result in zmm1.                                     |
| EVEX.128.66.MAP6.W0 B8 /r<br>VFMADD231PH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm2 and<br>xmm3/m128/m16bcst, add to xmm1, and store<br>the result in xmm1.                                     |
| EVEX.256.66.MAP6.W0 B8 /r<br>VFMADD231PH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm2 and<br>ymm3/m256/m16bcst, add to ymm1, and store<br>the result in ymm1.                                     |
| EVEX.512.66.MAP6.W0 B8 /r<br>VFMADD231PH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m16bcst {er}  | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm2 and<br>zmm3/m512/m16bcst, add to zmm1, and store<br>the result in zmm1.                                     |
| EVEX.128.66.MAP6.W0 9C /r<br>VFNMADD132PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm3/m128/m16bcst, and negate the value.<br>Add this value to xmm2, and store the result in<br>xmm1. |
| EVEX.256.66.MAP6.W0 9C /r<br>VFNMADD132PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm3/m256/m16bcst, and negate the value.<br>Add this value to ymm2, and store the result in<br>ymm1. |
| EVEX.512.66.MAP6.W0 9C /r<br>VFNMADD132PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and<br>zmm3/m512/m16bcst, and negate the value.<br>Add this value to zmm2, and store the result in<br>zmm1. |
| EVEX.128.66.MAP6.W0 AC /r<br>VFNMADD213PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm2, and negate the value. Add this value to<br>xmm3/m128/m16bcst, and store the result in<br>xmm1. |
| EVEX.256.66.MAP6.W0 AC /r<br>VFNMADD213PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm2, and negate the value. Add this value to<br>ymm3/m256/m16bcst, and store the result in<br>ymm1. |

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|-----------|------------------------------|-------------------------|--|
| EVEX.512.66.MAP6.WO AC /r<br>VFNMADD213PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and zmm2, and negate the value. Add this value to zmm3/m512/m16bcst, and store the result in zmm1. |
| EVEX.128.66.MAP6.WO BC /r<br>VFNMADD231PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, and negate the value. Add this value to xmm1, and store the result in xmm1. |
| EVEX.256.66.MAP6.WO BC /r<br>VFNMADD231PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, and negate the value. Add this value to ymm1, and store the result in ymm1. |
| EVEX.512.66.MAP6.WO BC /r<br>VFNMADD231PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, and negate the value. Add this value to zmm1, and store the result in zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1        | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|------------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (r, w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a packed multiply-add or negated multiply-add computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation “132”, “213” and “231” indicate the use of the operands in  $\pm A * B + C$ , where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-2.

The destination elements are updated according to the writemask.

**Table 5-2. VF[,N]MADD[132,213,231]PH Notation for Operands**

| Notation | Operands                        |
|----------|---------------------------------|
| 132      | $dest = \pm dest * src3 + src2$ |
| 231      | $dest = \pm src2 * src3 + dest$ |
| 213      | $dest = \pm src2 * dest + src3$ |

**Operation****VF[,N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-DEST.fp16[j]\*SRC3.fp16[j] + SRC2.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j]\*SRC3.fp16[j] + SRC2.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[,N]MADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] \* t3 + SRC2.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j] \* t3 + SRC2.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[,N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]\*DEST.fp16[j] + SRC3.fp16[j])

ELSE

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*DEST.fp16[j] + SRC3.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[,N]MADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] \* DEST.fp16[j] + t3 )

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* DEST.fp16[j] + t3 )

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]\*SRC3.fp16[j] + DEST.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*SRC3.fp16[j] + DEST.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[N]MADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] \* t3 + DEST.fp16[j] )

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* t3 + DEST.fp16[j] )

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VFMADD132PH, VFMADD213PH, and VFMADD231PH:

```

__m128h __mm_fmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h __mm_mask_fmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h __mm_mask3_fmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h __mm_maskz_fmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h __mm256_fmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h __mm256_mask_fmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h __mm256_mask3_fmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h __mm256_maskz_fmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h __mm512_fmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h __mm512_mask_fmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h __mm512_mask3_fmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h __mm512_maskz_fmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h __mm512_fmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h __mm512_mask_fmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h __mm512_mask3_fmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h __mm512_maskz_fmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

```

VFNMADD132PH, VFNMADD213PH, and VFNMADD231PH:

```

__m128h __mm_fnmadd_ph (__m128h a, __m128h b, __m128h c);
__m128h __mm_mask_fnmadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h __mm_mask3_fnmadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h __mm_maskz_fnmadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h __mm256_fnmadd_ph (__m256h a, __m256h b, __m256h c);
__m256h __mm256_mask_fnmadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h __mm256_mask3_fnmadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h __mm256_maskz_fnmadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h __mm512_fnmadd_ph (__m512h a, __m512h b, __m512h c);
__m512h __mm512_mask_fnmadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h __mm512_mask3_fnmadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h __mm512_maskz_fnmadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h __mm512_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h __mm512_mask_fnmadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h __mm512_mask3_fnmadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h __mm512_maskz_fnmadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal

**Other Exceptions**

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



## VFMADDRND231PD - Fused Multiply-Add of Packed Double-Precision Floating-Point Values with rounding control

| Opcode/<br>Instruction  | Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------------|--------------------------|--|
| VEX.DDS.128.66.0F3A.W1 B8 /r /ib<br>VFMADDRND231PD xmm0,<br>xmm1, xmm2/m128, imm8 | V/V             | FMA                      | Multiply packed double-precision floating-point values from xmm1 and xmm2/mem, add to xmm0 and put result in xmm0. |
| VEX.DDS.256.66.0F3A.W1 B8 /r /ib<br>VFMADDRND231PD ymm0,<br>ymm1, ymm2/m256, imm8 | V/V             | FMA                      | Multiply packed double-precision floating-point values from ymm1 and ymm2/mem, add to ymm0 and put result in ymm0. |

### Description

Multiplies the two or four 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, adds the infinite precision intermediate result to the two or four packed 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).

The immediate byte defines several bit fields that control rounding, DAZ, FTZ, and exception suppression (See Table 5-3). The rounding mode specified in MXCSR.RC may be bypassed if the immediate bit called MS1 (MXCSR.RC Override) is set. Likewise, the MXCSR.FTZ and MXCSR.DAZ may also be bypassed if the immediate bit called MS2 (MXCSR.FTZ/DAZ Override) is set. In case SAE (Suppress All Exceptions) bit is set (i.e. imm8[3] = 1), the status flags in MXCSR are not updated and no SIMD floating-point exceptions are raised. When SAE bit is not set (i.e. imm8[3] = 0) then SIMD floating-point exceptions are signaled according to the MXCSR. If any result operand is an SNaN then it will be converted to a QNaN.

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.

Table 5-3. Immediate Byte Encoding

| Bits           | Field Name/value | Description                                | Comment  |
|----------------|------------------|--|--|
| Imm8[1:0]<br>] | RC=0             | Round to nearest even                      | If Imm8[2] = 1   |
|                | RC=1             | Round down                                 |  |
|                | RC=2             | Round up                                   |  |
|                | RC=3             | Truncate                                   |  |
| Imm8[2]        | MS1=0            | Use MXCSR.RC for rounding                  | Ignore MXCSR.RC  |
|                | MS1=1            | Use Imm8[1:0] for rounding                 |  |
| Imm8[3]        | SAE=0            | Use MXCSR Exception Mask settings          | Numerical result is computed as if FP exceptions are masked. |
|                | SAE=1            | Suppress all Exception signaling           |  |
| Imm8[4]        | MS2=0            | Use MXCSR.DAZ and MXCSR.FTZ                | Ignore MXCSR.DAZ and MXCSR.FTZ                               |
|                | MS2=1            | Use Imm8[6:5] to control DAZ/FTZ operation |  |
| Imm8[5]        | DAZ              | Control DAZ                                | IF MS2 = 1   |
| Imm8[6]        | FTZ              | Control FTZ                                | IF MS2 = 1   |
| Imm8[7]        | MBZ              | Must be zero                               |  |

Compiler tools may optionally support the complementary mnemonic VMADDRND321PD. 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. See also Section 2.3.1, “FMA Instruction Operand Order and Arithmetic Behavior”

## Operation

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

### VFMADDRND231PD DEST, SRC2, SRC3, imm8

```

IF (VEX.128) THEN
    MAXVL = 2
ELSEIF (VEX.256)
    MAXVL = 4
FI
IF (imm8[3] = 1) THEN
    Suppress_SIMD_Exception_Signaling_Reporting();
FI
For i = 0 to MAXVL-1 {
    n = 64*i;
    DEST[n+63:n] ← RoundFPControl_Imm((SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n]), imm8)
}
IF (VEX.128) THEN
    DEST[255:128] ← 0
FI

```

```
IF (imm8[3] = 1) THEN  
    Resume_SIMD_Exception_Signaling_Reporting();  
FI
```

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

```
VFMADDRND231PD __m128d _mm_fmaddround_pd (__m128d a, __m128d b, __m128d c, const int ctrl);  
VFMADDRND231PD __m256d _mm256_fmaddround_pd (__m256d a, __m256d b, __m256d c, const int ctrl);
```

### SIMD Floating-Point Exceptions

```
IF imm[3] = 1 Then  
    None  
Else  
    Overflow, Underflow, Invalid, Precision, Denormal  
FI
```

### Other Exceptions

See Exceptions Type 2

## VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 98 /r<br>VFMADD132PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.          |
| VEX.128.66.0F38.W0 A8 /r<br>VFMADD213PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.          |
| VEX.128.66.0F38.W0 B8 /r<br>VFMADD231PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.          |
| VEX.256.66.0F38.W0 98 /r<br>VFMADD132PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.          |
| VEX.256.66.0F38.W0 A8 /r<br>VFMADD213PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.          |
| VEX.256.66.0F38.0 B8 /r<br>VFMADD231PS ymm1, ymm2,<br>ymm3/m256                       | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.          |
| EVEX.128.66.0F38.W0 98 /r<br>VFMADD132PS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1. |
| EVEX.128.66.0F38.W0 A8 /r<br>VFMADD213PS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m32bcst and put result in xmm1. |
| EVEX.128.66.0F38.W0 B8 /r<br>VFMADD231PS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and put result in xmm1. |
| EVEX.256.66.0F38.W0 98 /r<br>VFMADD132PS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1. |
| EVEX.256.66.0F38.W0 A8 /r<br>VFMADD213PS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m32bcst and put result in ymm1. |
| EVEX.256.66.0F38.W0 B8 /r<br>VFMADD231PS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and put result in ymm1. |
| EVEX.512.66.0F38.W0 98 /r<br>VFMADD132PS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1. |
| EVEX.512.66.0F38.W0 A8 /r<br>VFMADD213PS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m32bcst and put result in zmm1. |
| EVEX.512.66.0F38.W0 B8 /r<br>VFMADD231PS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1. |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## Description

Performs a set of SIMD multiply-add computation on packed single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

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

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

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

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

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

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

## Operation

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

**VFMADD132PS DEST, SRC2, SRC3**

```

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

```

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

**VFMAADD213PS 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
```

**VFMAADD231PS 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
```

**VFMAADD132PS 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

```

**VFMAADD213PS 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

```

**VFMAADD231PS 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

```

**VFMAADD231PS 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[j+31:i] :=
      RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] + DEST[j+31:i])
    ELSE
      DEST[j+31:i] :=
      RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] + DEST[j+31:i])
    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[j+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

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

```

VFMADDxxxPS __m512 __mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_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_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 __mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 __mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 __mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 __mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

```

### SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

### Other Exceptions

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

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

## VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.LIG.66.0F38.W1 99 /r<br>VFMADD132SD xmm1, xmm2,<br>xmm3/m64               | A         | V/V                          | FMA                      | Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1. |
| VEX.LIG.66.0F38.W1 A9 /r<br>VFMADD213SD xmm1, xmm2,<br>xmm3/m64               | A         | V/V                          | FMA                      | Multiply scalar double precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1. |
| VEX.LIG.66.0F38.W1 B9 /r<br>VFMADD231SD xmm1, xmm2,<br>xmm3/m64               | A         | V/V                          | FMA                      | Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W1 99 /r<br>VFMADD132SD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | B         | V/V                          | AVX512F                  | Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W1 A9 /r<br>VFMADD213SD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | B         | V/V                          | AVX512F                  | Multiply scalar double precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W1 B9 /r<br>VFMADD231SD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | B         | V/V                          | AVX512F                  | Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | N/A           | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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

```

**VFMAADD231SD 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

```

**VFMAADD132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] := MAXVL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0

```

**VFMAADD213SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0

```

**VFMAADD231SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] := RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:63] := DEST[127:63]
DEST[MAXVL-1:128] := 0

```

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

```

VFMAADDxxxSD __m128d __mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMAADDxxxSD __m128d __mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMAADDxxxSD __m128d __mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMAADDxxxSD __m128d __mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMAADDxxxSD __m128d __mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMAADDxxxSD __m128d __mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMAADDxxxSD __m128d __mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMAADDxxxSD __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.”

## VF[,N]MADD[132,213,231]SH—Fused Multiply-Add of Scalar FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.66.MAP6.W0 99 /r<br>VFMADD132SH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er}  | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm1 and xmm3/<br>m16, add to xmm2, and store the result in xmm1.                                     |
| EVEX.LLIG.66.MAP6.W0 A9 /r<br>VFMADD213SH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er}  | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm1 and xmm2, add<br>to xmm3/m16, and store the result in xmm1.                                      |
| EVEX.LLIG.66.MAP6.W0 B9 /r<br>VFMADD231SH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er}  | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm2 and xmm3/<br>m16, add to xmm1, and store the result in xmm1.                                     |
| EVEX.LLIG.66.MAP6.W0 9D /r<br>VFNMADD132SH xmm1{k1}{z},<br>xmm2, xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm1 and xmm3/<br>m16, and negate the value. Add this value to<br>xmm2, and store the result in xmm1. |
| EVEX.LLIG.66.MAP6.W0 AD /r<br>VFNMADD213SH xmm1{k1}{z},<br>xmm2, xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm1 and xmm2, and<br>negate the value. Add this value to xmm3/m16,<br>and store the result in xmm1.  |
| EVEX.LLIG.66.MAP6.W0 BD /r<br>VFNMADD231SH xmm1{k1}{z},<br>xmm2, xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm2 and xmm3/<br>m16, and negate the value. Add this value to<br>xmm1, and store the result in xmm1. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1        | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|------------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (r, w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Performs a scalar multiply-add or negated multiply-add computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction add the negated infinite precision intermediate product to the corresponding remaining operand. The notation “132”, “213” and “231” indicate the use of the operands in  $\pm A * B + C$ , where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-4.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

**Table 5-4. VF[,N]MADD[132,213,231]SH Notation for Operands**

| Notation | Operands                        |
|----------|---------------------------------|
| 132      | $dest = \pm dest * src3 + src2$ |
| 231      | $dest = \pm src2 * src3 + dest$ |
| 213      | $dest = \pm src2 * dest + src3$ |

**Operation****VF[,N]MADD132SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

IF k1[0] OR \*no writemask\*:

IF \*negative form\*:

DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]\*SRC3.fp16[0] + SRC2.fp16[0])

ELSE:

DEST.fp16[0] := RoundFPControl(DEST.fp16[0]\*SRC3.fp16[0] + SRC2.fp16[0])

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged

DEST[MAXVL-1:128] := 0

**VF[,N]MADD213SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

IF k1[0] OR \*no writemask\*:

IF \*negative form\*:

DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]\*DEST.fp16[0] + SRC3.fp16[0])

ELSE:

DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]\*DEST.fp16[0] + SRC3.fp16[0])

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged

DEST[MAXVL-1:128] := 0

**VF[,N]MADD231SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

IF k1[0] OR \*no writemask\*:

IF \*negative form\*:

DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]\*SRC3.fp16[0] + DEST.fp16[0])

ELSE:

DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]\*SRC3.fp16[0] + DEST.fp16[0])

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged

DEST[MAXVL-1:128] := 0

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

VFMADD132SH, VFMADD213SH, and VFMADD231SH:

```
__m128h _mm_fmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmadd_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

VFMADD132SH, VFMADD213SH, and VFMADD231SH:

```
__m128h _mm_fmadd_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmadd_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmadd_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmadd_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmadd_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmadd_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmadd_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmadd_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal

**Other Exceptions**

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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| VEX.LIG.66.0F38.W0 99 /r<br>VFMADD132SS xmm1, xmm2,<br>xmm3/m32               | A          | V/V                          | FMA                      | Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1. |
| VEX.LIG.66.0F38.W0 A9 /r<br>VFMADD213SS xmm1, xmm2,<br>xmm3/m32               | A          | V/V                          | FMA                      | Multiply scalar single precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1. |
| VEX.LIG.66.0F38.W0 B9 /r<br>VFMADD231SS xmm1, xmm2,<br>xmm3/m32               | A          | V/V                          | FMA                      | Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W0 99 /r<br>VFMADD132SS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | B          | V/V                          | AVX512F                  | Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W0 A9 /r<br>VFMADD213SS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | B          | V/V                          | AVX512F                  | Multiply scalar single precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W0 B9 /r<br>VFMADD231SS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | B          | V/V                          | AVX512F                  | Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | N/A           | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Performs a SIMD multiply-add computation on single precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

**VFMADD132SS:** Multiplies the low single precision floating-point value from the first source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the second source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

**VFMADD213SS:** Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the third source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

**VFMADD231SS:** Multiplies the low single precision floating-point value from the second source operand to the low single precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single precision floating-point value in the first source operand, performs rounding and stores the resulting single precision floating-point value to the destination operand (first source operand).

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



EVEX encoded version: The low doubleword element of the destination is updated according to the writemask. Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NaNs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

### Operation

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

#### VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] := RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] := 0
    FI;
FI;
DEST[127:32] := DEST[127:32]
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[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0
```

**VFMAADD231SS 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[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

**VFMAADD132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] := RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

**VFMAADD213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

**VFMAADD231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] := RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

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

```

VFMAADDxxxSS __m128 __mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMAADDxxxSS __m128 __mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMAADDxxxSS __m128 __mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMAADDxxxSS __m128 __mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMAADDxxxSS __m128 __mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMAADDxxxSS __m128 __mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMAADDxxxSS __m128 __mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMAADDxxxSS __m128 __mm_fmadd_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."

## VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W1 96 /r<br>VFMADDSUB132PD xmm1, xmm2,<br>xmm3/m128                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.                                  |
| VEX.128.66.0F38.W1 A6 /r<br>VFMADDSUB213PD xmm1, xmm2,<br>xmm3/m128                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.                                  |
| VEX.128.66.0F38.W1 B6 /r<br>VFMADDSUB231PD xmm1, xmm2,<br>xmm3/m128                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.                                  |
| VEX.256.66.0F38.W1 96 /r<br>VFMADDSUB132PD ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.                                  |
| VEX.256.66.0F38.W1 A6 /r<br>VFMADDSUB213PD ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.                                  |
| VEX.256.66.0F38.W1 B6 /r<br>VFMADDSUB231PD ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.                                  |
| EVEX.128.66.0F38.W1 A6 /r<br>VFMADDSUB213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1. |
| EVEX.128.66.0F38.W1 B6 /r<br>VFMADDSUB231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1. |
| EVEX.128.66.0F38.W1 96 /r<br>VFMADDSUB132PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1. |
| EVEX.256.66.0F38.W1 A6 /r<br>VFMADDSUB213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1. |
| EVEX.256.66.0F38.W1 B6 /r<br>VFMADDSUB231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1. |
| EVEX.256.66.0F38.W1 96 /r<br>VFMADDSUB132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1. |

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W1 A6 /r<br>VFMADDSUB213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B          | V/V                          | AVX512F                  | 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. |
| EVEX.512.66.0F38.W1 B6 /r<br>VFMADDSUB231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B          | V/V                          | AVX512F                  | 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. |
| EVEX.512.66.0F38.W1 96 /r<br>VFMADDSUB132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B          | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

#### Description

VFMADDSUB132PD: Multiplies the two, four, or eight packed double precision floating-point values from the first source operand to the two or four packed double precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double precision floating-point elements and subtracts the even double precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double precision floating-point values to the destination operand (first source operand).

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

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

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

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

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

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

### Operation

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

#### VFMADDSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] := RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[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])
    DEST[191:128] := RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192])
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])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192])
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])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192])
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[63:0] - SRC2[i+63:i])
            ELSE
              DEST[i+63:i] :=
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[i+63:i] :=
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
            ELSE
              DEST[i+63:i] :=
                RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
          FI;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

```

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

```

**VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

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

```

**VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*

```



```

THEN
  IF (EVEX.b = 1)
    THEN
      DEST[j+63:i] :=
        RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[63:0] - DEST[j+63:i])
    ELSE
      DEST[j+63:i] :=
        RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[j+63:i] - DEST[j+63:i])
    FI;
  ELSE
    IF (EVEX.b = 1)
      THEN
        DEST[j+63:i] :=
          RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[63:0] + DEST[j+63:i])
      ELSE
        DEST[j+63:i] :=
          RoundFPControl_MXCSR(SRC2[j+63:i]*SRC3[j+63:i] + DEST[j+63:i])
      FI;
    FI
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[j+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[j+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

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

```

VFMADDSUBxxxPD __m512d __mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d __mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d __mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d __mm512_mask3_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDSUBxxxPD __m256d __mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d __mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_mask_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d __mm_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxxxPD __m128d __mm_fmaddsub_pd(__m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m256d __mm256_fmaddsub_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.”

## VFMADDSUB132PH/VFMADDSUB213PH/VFMADDSUB231PH—Fused Multiply-Alternating Add/Subtract of Packed FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP6.W0 96 /r<br>VFMADDSUB132PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm3/m128/m16bcst, add/subtract elements in<br>xmm2, and store the result in xmm1 subject to<br>writemask k1.  |
| EVEX.256.66.MAP6.W0 96 /r<br>VFMADDSUB132PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm3/m256/m16bcst, add/subtract elements in<br>ymm2, and store the result in ymm1 subject to<br>writemask k1.  |
| EVEX.512.66.MAP6.W0 96 /r<br>VFMADDSUB132PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and<br>zmm3/m512/m16bcst, add/subtract elements in<br>zmm2, and store the result in zmm1 subject to<br>writemask k1.  |
| EVEX.128.66.MAP6.W0 A6 /r<br>VFMADDSUB213PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm2, add/subtract elements in xmm3/m128/<br>m16bcst, and store the result in xmm1 subject to<br>writemask k1. |
| EVEX.256.66.MAP6.W0 A6 /r<br>VFMADDSUB213PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm2, add/subtract elements in ymm3/m256/<br>m16bcst, and store the result in ymm1 subject to<br>writemask k1. |
| EVEX.512.66.MAP6.W0 A6 /r<br>VFMADDSUB213PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and<br>zmm2, add/subtract elements in zmm3/m512/<br>m16bcst, and store the result in zmm1 subject to<br>writemask k1. |
| EVEX.128.66.MAP6.W0 B6 /r<br>VFMADDSUB231PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm2 and<br>xmm3/m128/m16bcst, add/subtract elements in<br>xmm1, and store the result in xmm1 subject to<br>writemask k1.  |
| EVEX.256.66.MAP6.W0 B6 /r<br>VFMADDSUB231PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm2 and<br>ymm3/m256/m16bcst, add/subtract elements in<br>ymm1, and store the result in ymm1 subject to<br>writemask k1.  |
| EVEX.512.66.MAP6.W0 B6 /r<br>VFMADDSUB231PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm2 and<br>zmm3/m512/m16bcst, add/subtract elements in<br>zmm1, and store the result in zmm1 subject to<br>writemask k1.  |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1        | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|------------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (r, w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a packed multiply-add (odd elements) or multiply-subtract (even elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation “132”, “213” and “231” indicate the use of the operands in  $A * B \pm C$ , where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-8. The destination elements are updated according to the writemask.

**Table 5-5. VFMADDSUB[132,213,231]PH Notation for Odd and Even Elements**

| Notation | Odd Elements                | Even Elements               |
|----------|-----------------------------|-----------------------------|
| 132      | $dest = dest * src3 + src2$ | $dest = dest * src3 - src2$ |
| 231      | $dest = src2 * src3 + dest$ | $dest = src2 * src3 - dest$ |
| 213      | $dest = src2 * dest + src3$ | $dest = src2 * dest - src3$ |

**Operation****VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j] \* SRC3.fp16[j] - SRC2.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j] \* SRC3.fp16[j] + SRC2.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VFMADDSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j] \* t3 - SRC2.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j] \* t3 + SRC2.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*DEST.fp16[j] - SRC3.fp16[j])

ELSE

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*DEST.fp16[j] + SRC3.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VFMADDSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* DEST.fp16[j] - t3)

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* DEST.fp16[j] + t3)

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* SRC3.fp16[j] - DEST.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* SRC3.fp16[j] + DEST.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VFMADDSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* t3 - DEST.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* t3 + DEST.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VFMADDSUB132PH, VFMADDSUB213PH, and VFMADDSUB231PH:

```

__m128h _mm_fmaddsub_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmaddsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmaddsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmaddsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmaddsub_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmaddsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmaddsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmaddsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmaddsub_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmaddsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmaddsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmaddsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmaddsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmaddsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmaddsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 96 /r<br>VFMADDSUB132PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.                                  |
| VEX.128.66.0F38.W0 A6 /r<br>VFMADDSUB213PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.                                  |
| VEX.128.66.0F38.W0 B6 /r<br>VFMADDSUB231PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.                                  |
| VEX.256.66.0F38.W0 96 /r<br>VFMADDSUB132PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.                                  |
| VEX.256.66.0F38.W0 A6 /r<br>VFMADDSUB213PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.                                  |
| VEX.256.66.0F38.W0 B6 /r<br>VFMADDSUB231PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.                                  |
| EVEX.128.66.0F38.W0 A6 /r<br>VFMADDSUB213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1. |
| EVEX.128.66.0F38.W0 B6 /r<br>VFMADDSUB231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1. |
| EVEX.128.66.0F38.W0 96 /r<br>VFMADDSUB132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, add/subtract elements in zmm2 and put result in xmm1 subject to writemask k1. |
| EVEX.256.66.0F38.W0 A6 /r<br>VFMADDSUB213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1. |
| EVEX.256.66.0F38.W0 B6 /r<br>VFMADDSUB231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1. |
| EVEX.256.66.0F38.W0 96 /r<br>VFMADDSUB132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1. |
| EVEX.512.66.0F38.W0 A6 /r<br>VFMADDSUB213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1. |
| EVEX.512.66.0F38.W0 B6 /r<br>VFMADDSUB231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1. |
| EVEX.512.66.0F38.W0 96 /r<br>VFMADDSUB132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1. |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## Description

VFMADDSUB132PS: Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single precision floating-point elements and subtracts the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

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

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

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])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}
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])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}
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])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}
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[j+31:i] - SRC2[j+31:i])

```

```

        ELSE DEST[j+31:i] :=
            RoundFPControl(DEST[j+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
    FI
ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
        DEST[j+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[j+31:i] :=
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[31:0] - SRC2[j+31:i])
                        ELSE
                            DEST[j+31:i] :=
                                RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
                        FI;
                    ELSE
                        IF (EVEX.b = 1)
                            THEN
                                DEST[j+31:i] :=
                                    RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[31:0] + SRC2[j+31:i])
                            ELSE
                                DEST[j+31:i] :=
                                    RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
                            FI;
                        FI;
                    ELSE
                        IF (EVEX.b = 1)
                            THEN
                                DEST[j+31:i] :=
                                    RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[31:0] + SRC2[j+31:i])
                            ELSE
                                DEST[j+31:i] :=
                                    RoundFPControl_MXCSR(DEST[j+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
                            FI;
                        FI;
                    ELSE
                        IF *merging-masking*           ; merging-masking
                            THEN *DEST[j+31:i] remains unchanged*
                        ELSE                             ; zeroing-masking
                            DEST[j+31:i] := 0
                        FI
                    FI;
                ELSE
                    IF *merging-masking*           ; merging-masking
                        THEN *DEST[j+31:i] remains unchanged*
                    ELSE                             ; zeroing-masking
                        DEST[j+31:i] := 0
                    FI
                FI;
            ELSE
                IF *merging-masking*           ; merging-masking
                    THEN *DEST[j+31:i] remains unchanged*
                ELSE                             ; zeroing-masking
                    DEST[j+31:i] := 0
                FI
            FI;
        ELSE
            IF *merging-masking*           ; merging-masking
                THEN *DEST[j+31:i] remains unchanged*
            ELSE                             ; zeroing-masking
                DEST[j+31:i] := 0
            FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] := 0

```

**VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 32

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

THEN

IF j \*is even\*

THEN DEST[i+31:i] :=

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

ELSE DEST[i+31:i] :=

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j := 0 TO KL-1

i := j \* 32

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

THEN

IF 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[31:0])

ELSE

DEST[i+31:i] :=

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI;

```

        FI
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[i+31:i] := 0
        FI
    FI;
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;
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[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
    FI;
ELSE
    IF (EVEX.b = 1)
        THEN
            DEST[j+31:i] :=
                RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] + DEST[j+31:i])
        ELSE
            DEST[j+31:i] :=
                RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] + DEST[j+31:i])
    FI;
FI
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[j+31:i] := 0
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

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

```

VFMADDSUBxxxPS __m512_mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512_mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512_mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512_mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512_mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512_mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512_mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512_mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxxPS __m256_mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256_mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256_mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128_mm_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128_mm_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128_mm_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m128_mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __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.”

## VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W1 9A /r<br>VFMSUB132PD xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.                                  |
| VEX.128.66.0F38.W1 AA /r<br>VFMSUB213PD xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.                                  |
| VEX.128.66.0F38.W1 BA /r<br>VFMSUB231PD xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.                                  |
| VEX.256.66.0F38.W1 9A /r<br>VFMSUB132PD ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.                                  |
| VEX.256.66.0F38.W1 AA /r<br>VFMSUB213PD ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.                                  |
| VEX.256.66.0F38.W1 BA /r<br>VFMSUB231PD ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.S                                 |
| EVEX.128.66.0F38.W1 9A /r<br>VFMSUB132PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1. |
| EVEX.128.66.0F38.W1 AA /r<br>VFMSUB213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1. |
| EVEX.128.66.0F38.W1 BA /r<br>VFMSUB231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract xmm1 and put result in xmm1 subject to writemask k1. |
| EVEX.256.66.0F38.W1 9A /r<br>VFMSUB132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1. |
| EVEX.256.66.0F38.W1 AA /r<br>VFMSUB213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1. |
| EVEX.256.66.0F38.W1 BA /r<br>VFMSUB231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract ymm1 and put result in ymm1 subject to writemask k1. |
| EVEX.512.66.0F38.W1 9A /r<br>VFMSUB132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed double precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract zmm2 and put result in zmm1 subject to writemask k1. |
| EVEX.512.66.0F38.W1 AA /r<br>VFMSUB213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed double precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1. |
| EVEX.512.66.0F38.W1 BA /r<br>VFMSUB231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed double precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1. |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## Description

Performs a set of SIMD multiply-subtract computation on packed double precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

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

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

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

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

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

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

## Operation

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

**VFMSUB132PD DEST, SRC2, SRC3 (VEX encoded versions)**

```

IF (VEX.128) THEN
    MAXNUM := 2
ELSEIF (VEX.256)
    MAXNUM := 4
FI
For i = 0 to MAXNUM-1 {
    n := 64*i;
    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

```

**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;
    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

```

**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;
    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

```

**VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

```

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

```



```

        FI
    FI;
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;
    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, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d __mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d __mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d __mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d __mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d __mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d __mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d __mm_mask_fmsub_pd(__m128d a, __mmask8 k, __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 __m256d __mm256_fmsub_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.”

## VF[,N]MSUB[132,213,231]PH—Fused Multiply-Subtract of Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP6.WO 9A /r<br>VFMSUB132PH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm3/m128/m16bcst, subtract xmm2, and store<br>the result in xmm1 subject to writemask k1.   |
| EVEX.256.66.MAP6.WO 9A /r<br>VFMSUB132PH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm3/m256/m16bcst, subtract ymm2, and store<br>the result in ymm1 subject to writemask k1.   |
| EVEX.512.66.MAP6.WO 9A /r<br>VFMSUB132PH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and<br>zmm3/m512/m16bcst, subtract zmm2, and store<br>the result in zmm1 subject to writemask k1.   |
| EVEX.128.66.MAP6.WO AA /r<br>VFMSUB213PH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm2, subtract xmm3/m128/m16bcst, and store<br>the result in xmm1 subject to writemask k1.   |
| EVEX.256.66.MAP6.WO AA /r<br>VFMSUB213PH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm2, subtract ymm3/m256/m16bcst, and store<br>the result in ymm1 subject to writemask k1.   |
| EVEX.512.66.MAP6.WO AA /r<br>VFMSUB213PH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and<br>zmm2, subtract zmm3/m512/m16bcst, and store<br>the result in zmm1 subject to writemask k1.   |
| EVEX.128.66.MAP6.WO BA /r<br>VFMSUB231PH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm2 and<br>xmm3/m128/m16bcst, subtract xmm1, and store<br>the result in xmm1 subject to writemask k1.   |
| EVEX.256.66.MAP6.WO BA /r<br>VFMSUB231PH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm2 and<br>ymm3/m256/m16bcst, subtract ymm1, and store<br>the result in ymm1 subject to writemask k1.   |
| EVEX.512.66.MAP6.WO BA /r<br>VFMSUB231PH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm2 and<br>zmm3/m512/m16bcst, subtract zmm1, and store<br>the result in zmm1 subject to writemask k1.   |
| EVEX.128.66.MAP6.WO 9E /r<br>VFMSUB132PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm3/m128/m16bcst, and negate the value.<br>Subtract xmm2 from this value, and store the<br>result in xmm1 subject to writemask k1.  |
| EVEX.256.66.MAP6.WO 9E /r<br>VFMSUB132PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm3/m256/m16bcst, and negate the value.<br>Subtract ymm2 from this value, and store the<br>result in ymm1 subject to writemask k1.  |
| EVEX.512.66.MAP6.WO 9E /r<br>VFMSUB132PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and<br>zmm3/m512/m16bcst, and negate the value.<br>Subtract zmm2 from this value, and store the<br>result in zmm1 subject to writemask k1.  |
| EVEX.128.66.MAP6.WO AE /r<br>VFMSUB213PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and<br>xmm2, and negate the value. Subtract xmm3/<br>m128/m16bcst from this value, and store the<br>result in xmm1 subject to writemask k1. |
| EVEX.256.66.MAP6.WO AE /r<br>VFMSUB213PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and<br>ymm2, and negate the value. Subtract ymm3/<br>m256/m16bcst from this value, and store the<br>result in ymm1 subject to writemask k1. |

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.512.66.MAP6.W0 AE /r<br>VFNMSUB213PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and zmm2, and negate the value. Subtract zmm3/m512/m16bcst from this value, and store the result in zmm1 subject to writemask k1. |
| EVEX.128.66.MAP6.W0 BE /r<br>VFNMSUB231PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, and negate the value. Subtract xmm1 from this value, and store the result in xmm1 subject to writemask k1. |
| EVEX.256.66.MAP6.W0 BE /r<br>VFNMSUB231PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, and negate the value. Subtract ymm1 from this value, and store the result in ymm1 subject to writemask k1. |
| EVEX.512.66.MAP6.W0 BE /r<br>VFNMSUB231PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, and negate the value. Subtract zmm1 from this value, and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1        | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|------------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (r, w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a packed multiply-subtract or a negated multiply-subtract computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The "N" (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation "132", "213" and "231" indicate the use of the operands in  $\pm A * B - C$ , where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-6.

The destination elements are updated according to the writemask.

**Table 5-6. VF[,N]MSUB[132,213,231]PH Notation for Operands**

| Notation | Operands                    |
|----------|-----------------------------|
| 132      | dest = $\pm$ dest*src3-src2 |
| 231      | dest = $\pm$ src2*src3-dest |
| 213      | dest = $\pm$ src2*dest-src3 |

**Operation****VF[,N]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-DEST.fp16[j]\*SRC3.fp16[j] - SRC2.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j]\*SRC3.fp16[j] - SRC2.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[,N]MSUB132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-DEST.fp16[j] \* t3 - SRC2.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j] \* t3 - SRC2.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[N]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]\*DEST.fp16[j] - SRC3.fp16[j])

ELSE

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*DEST.fp16[j] - SRC3.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[N]MSUB213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] \* DEST.fp16[j] - t3 )

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* DEST.fp16[j] - t3 )

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[,N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j]\*SRC3.fp16[j] - DEST.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*SRC3.fp16[j] - DEST.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VF[,N]MSUB231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*negative form\*:

DEST.fp16[j] := RoundFPControl(-SRC2.fp16[j] \* t3 - DEST.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* t3 - DEST.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0



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

VFMSUB132PH, VFMSUB213PH, and VFMSUB231PH:

```

__m128h __mm_fmsub_ph (__m128h a, __m128h b, __m128h c);
__m128h __mm_mask_fmsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h __mm_mask3_fmsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h __mm_maskz_fmsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h __mm256_fmsub_ph (__m256h a, __m256h b, __m256h c);
__m256h __mm256_mask_fmsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h __mm256_mask3_fmsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h __mm256_maskz_fmsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h __mm512_fmsub_ph (__m512h a, __m512h b, __m512h c);
__m512h __mm512_mask_fmsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h __mm512_mask3_fmsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h __mm512_maskz_fmsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h __mm512_fmsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h __mm512_mask_fmsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h __mm512_mask3_fmsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h __mm512_maskz_fmsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

```

VFNMSUB132PH, VFNMSUB213PH, and VFNMSUB231PH:

```

__m128h __mm_fnmsub_ph (__m128h a, __m128h b, __m128h c);
__m128h __mm_mask_fnmsub_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h __mm_mask3_fnmsub_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h __mm_maskz_fnmsub_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h __mm256_fnmsub_ph (__m256h a, __m256h b, __m256h c);
__m256h __mm256_mask_fnmsub_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h __mm256_mask3_fnmsub_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h __mm256_maskz_fnmsub_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h __mm512_fnmsub_ph (__m512h a, __m512h b, __m512h c);
__m512h __mm512_mask_fnmsub_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h __mm512_mask3_fnmsub_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h __mm512_maskz_fnmsub_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h __mm512_fnmsub_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h __mm512_mask_fnmsub_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h __mm512_mask3_fnmsub_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h __mm512_maskz_fnmsub_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

```

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

### Other Exceptions

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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 9A /r<br>VFMSUB132PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.          |
| VEX.128.66.0F38.W0 AA /r<br>VFMSUB213PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.          |
| VEX.128.66.0F38.W0 BA /r<br>VFMSUB231PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.          |
| VEX.256.66.0F38.W0 9A /r<br>VFMSUB132PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.          |
| VEX.256.66.0F38.W0 AA /r<br>VFMSUB213PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.          |
| VEX.256.66.0F38.W0 BA /r<br>VFMSUB231PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.          |
| EVEX.128.66.0F38.W0 9A /r<br>VFMSUB132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1. |
| EVEX.128.66.0F38.W0 AA /r<br>VFMSUB213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1. |
| EVEX.128.66.0F38.W0 BA /r<br>VFMSUB231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1. |
| EVEX.256.66.0F38.W0 9A /r<br>VFMSUB132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1. |
| EVEX.256.66.0F38.W0 AA /r<br>VFMSUB213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1. |
| EVEX.256.66.0F38.W0 BA /r<br>VFMSUB231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | Multiply packed single precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1. |
| EVEX.512.66.0F38.W0 9A /r<br>VFMSUB132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1. |
| EVEX.512.66.0F38.W0 AA /r<br>VFMSUB213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1. |
| EVEX.512.66.0F38.W0 BA /r<br>VFMSUB231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1. |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## Description

Performs a set of SIMD multiply-subtract computation on packed single precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

**VFMSUB132PS:** Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

**VFMSUB213PS:** Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the four, eight or sixteen packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

**VFMSUB231PS:** Multiplies the four, eight or sixteen packed single precision floating-point values from the second source to the four, eight or sixteen packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single precision floating-point values to the destination operand (first source operand).

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

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

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

## Operation

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

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

```

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

```

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

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

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

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

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

**VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
    i := j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] :=
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] := 0
```

```

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

```

**VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

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

```

**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[j+31:i] :=
      RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] - DEST[j+31:i])
    ELSE
      DEST[j+31:i] :=
      RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
    FI;
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[j+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

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

```

VFMSUBxxxPS __m512 __mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 __mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 __mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBxxxPS __m256 __mm256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 __mm256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 __mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m128 __mm_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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| VEX.LIG.66.0F38.W1 9B /r<br>VFMSUB132SD xmm1, xmm2,<br>xmm3/m64               | A          | V/V                          | FMA                      | Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1. |
| VEX.LIG.66.0F38.W1 AB /r<br>VFMSUB213SD xmm1, xmm2,<br>xmm3/m64               | A          | V/V                          | FMA                      | Multiply scalar double precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1. |
| VEX.LIG.66.0F38.W1 BB /r<br>VFMSUB231SD xmm1, xmm2,<br>xmm3/m64               | A          | V/V                          | FMA                      | Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W1 9B /r<br>VFMSUB132SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | B          | V/V                          | AVX512F                  | Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W1 AB /r<br>VFMSUB213SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | B          | V/V                          | AVX512F                  | Multiply scalar double precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W1 BB /r<br>VFMSUB231SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | B          | V/V                          | AVX512F                  | Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | N/A           | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Performs a SIMD multiply-subtract computation on the low packed double precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

**VFMSUB132SD:** Multiplies the low packed double precision floating-point value from the first source operand to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point values in the second source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

**VFMSUB213SD:** Multiplies the low packed double precision floating-point value from the second source operand to the low packed double precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the third source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).

**VFMSUB231SD:** Multiplies the low packed double precision floating-point value from the second source to the low packed double precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double precision floating-point value in the first source operand, performs rounding and stores the resulting packed double precision floating-point value to the destination operand (first source operand).



VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

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

## Operation

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

### VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```

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

```

### VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```

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

```

**VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

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

```

**VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[63:0] := RoundFPControl_MXCSR(DEST[63:0]*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, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d __mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d __mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d __mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d __mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d __mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

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

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

## VF[,N]MSUB[132,213,231]SH—Fused Multiply-Subtract of Scalar FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|---|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.66.MAP6.W0 9B /r<br>VFMSUB132SH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm1 and xmm3/<br>m16, subtract xmm2, and store the result in<br>xmm1 subject to writemask k1.  |
| EVEX.LLIG.66.MAP6.W0 AB /r<br>VFMSUB213SH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm1 and xmm2,<br>subtract xmm3/m16, and store the result in<br>xmm1 subject to writemask k1.   |
| EVEX.LLIG.66.MAP6.W0 BB /r<br>VFMSUB231SH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm2 and xmm3/<br>m16, subtract xmm1, and store the result in<br>xmm1 subject to writemask k1.  |
| EVEX.LLIG.66.MAP6.W0 9F /r<br>VFMSUB132SH xmm1{k1}{z},<br>xmm2, xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm1 and xmm3/<br>m16, and negate the value. Subtract xmm2 from<br>this value, and store the result in xmm1 subject<br>to writemask k1. |
| EVEX.LLIG.66.MAP6.W0 AF /r<br>VFMSUB213SH xmm1{k1}{z},<br>xmm2, xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm1 and xmm2, and<br>negate the value. Subtract xmm3/m16 from this<br>value, and store the result in xmm1 subject to<br>writemask k1.  |
| EVEX.LLIG.66.MAP6.W0 BF /r<br>VFMSUB231SH xmm1{k1}{z},<br>xmm2, xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply FP16 values from xmm2 and xmm3/<br>m16, and negate the value. Subtract xmm1 from<br>this value, and store the result in xmm1 subject<br>to writemask k1. |

## Instruction Operand Encoding

| Op/En | Tuple  | Operand 1        | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|------------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (r, w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## Description

This instruction performs a scalar multiply-subtract or negated multiply-subtract computation on the low FP16 values using three source operands and writes the result in the destination operand. The destination operand is also the first source operand. The “N” (negated) forms of this instruction subtract the remaining operand from the negated infinite precision intermediate product. The notation “132”, “213” and “231” indicate the use of the operands in  $\pm A * B - C$ , where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-7.

Bits 127:16 of the destination operand are preserved. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Table 5-7. VF[,N]MSUB[132,213,231]SH Notation for Operands

| Notation | Operands                        |
|----------|---------------------------------|
| 132      | $dest = \pm dest * src3 - src2$ |
| 231      | $dest = \pm src2 * src3 - dest$ |
| 213      | $dest = \pm src2 * dest - src3$ |

**Operation****VF[,N]MSUB132SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

IF k1[0] OR \*no writemask\*:

IF \*negative form\*:

DEST.fp16[0] := RoundFPControl(-DEST.fp16[0]\*SRC3.fp16[0] - SRC2.fp16[0])

ELSE:

DEST.fp16[0] := RoundFPControl(DEST.fp16[0]\*SRC3.fp16[0] - SRC2.fp16[0])

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged

DEST[MAXVL-1:128] := 0

**VF[,N]MSUB213SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

IF k1[0] OR \*no writemask\*:

IF \*negative form\*:

DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]\*DEST.fp16[0] - SRC3.fp16[0])

ELSE:

DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]\*DEST.fp16[0] - SRC3.fp16[0])

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged

DEST[MAXVL-1:128] := 0

**VF[,N]MSUB231SH DEST, SRC2, SRC3 (EVEX encoded versions)**

IF EVEX.b = 1 and SRC3 is a register:

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

IF k1[0] OR \*no writemask\*:

IF \*negative form\*:

DEST.fp16[0] := RoundFPControl(-SRC2.fp16[0]\*SRC3.fp16[0] - DEST.fp16[0])

ELSE:

DEST.fp16[0] := RoundFPControl(SRC2.fp16[0]\*SRC3.fp16[0] - DEST.fp16[0])

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

// else DEST.fp16[0] remains unchanged

//DEST[127:16] remains unchanged

DEST[MAXVL-1:128] := 0

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

VFMSUB132SH, VFMSUB213SH, and VFMSUB231SH:

```
__m128h _mm_fmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

VFNMSUB132SH, VFNMSUB213SH, and VFNMSUB231SH:

```
__m128h _mm_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask_fnmsub_round_sh (__m128h a, __mmask8 k, __m128h b, __m128h c, const int rounding);
__m128h _mm_mask3_fnmsub_round_sh (__m128h a, __m128h b, __m128h c, __mmask8 k, const int rounding);
__m128h _mm_maskz_fnmsub_round_sh (__mmask8 k, __m128h a, __m128h b, __m128h c, const int rounding);
__m128h _mm_fnmsub_sh (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fnmsub_sh (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fnmsub_sh (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fnmsub_sh (__mmask8 k, __m128h a, __m128h b, __m128h c);
```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal

**Other Exceptions**

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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| VEX.LIG.66.0F38.W0 9B /r<br>VFMSUB132SS xmm1, xmm2,<br>xmm3/m32               | A          | V/V                          | FMA                      | Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1. |
| VEX.LIG.66.0F38.W0 AB /r<br>VFMSUB213SS xmm1, xmm2,<br>xmm3/m32               | A          | V/V                          | FMA                      | Multiply scalar single precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1. |
| VEX.LIG.66.0F38.W0 BB /r<br>VFMSUB231SS xmm1, xmm2,<br>xmm3/m32               | A          | V/V                          | FMA                      | Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W0 9B /r<br>VFMSUB132SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | B          | V/V                          | AVX512F                  | Multiply scalar single precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W0 AB /r<br>VFMSUB213SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | B          | V/V                          | AVX512F                  | Multiply scalar single precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W0 BB /r<br>VFMSUB231SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | B          | V/V                          | AVX512F                  | Multiply scalar single precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | N/A           | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Performs a SIMD multiply-subtract computation on the low packed single precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

**VFMSUB132SS:** Multiplies the low packed single precision floating-point value from the first source operand to the low packed single precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point value to the destination operand (first source operand).

**VFMSUB213SS:** Multiplies the low packed single precision floating-point value from the second source operand to the low packed single precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts 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).

**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[127:32] := DEST[127:32]
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[127:32] := DEST[127:32]
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[127:32] := DEST[127:32]
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[127:32] := DEST[127:32]
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[127:32] := DEST[127:32]
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[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

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

```

VFMSUBxxxSS __m128 __mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 __mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 __mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 __mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 __mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 __mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 __mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 __mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

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

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



## VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double Precision Floating-Point Values

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W1 97 /r<br>VFMSUBADD132PD xmm1, xmm2,<br>xmm3/m128                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.                                  |
| VEX.128.66.0F38.W1 A7 /r<br>VFMSUBADD213PD xmm1, xmm2,<br>xmm3/m128                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.                                  |
| VEX.128.66.0F38.W1 B7 /r<br>VFMSUBADD231PD xmm1, xmm2,<br>xmm3/m128                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.                                  |
| VEX.256.66.0F38.W1 97 /r<br>VFMSUBADD132PD ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.                                  |
| VEX.256.66.0F38.W1 A7 /r<br>VFMSUBADD213PD ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.                                  |
| VEX.256.66.0F38.W1 B7 /r<br>VFMSUBADD231PD ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.                                  |
| EVEX.128.66.0F38.W1 97 /r<br>VFMSUBADD132PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W1 A7 /r<br>VFMSUBADD213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W1 B7 /r<br>VFMSUBADD231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W1 97 /r<br>VFMSUBADD132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W1 A7 /r<br>VFMSUBADD213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Multiply packed double precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1. |
| EVEX.256.66.0F38.W1 B7 /r<br>VFMSUBADD231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W1 97 /r<br>VFMSUBADD132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B          | V/V                          | AVX512F                  | 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. |
| EVEX.512.66.0F38.W1 A7 /r<br>VFMSUBADD213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B          | V/V                          | AVX512F                  | 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. |
| EVEX.512.66.0F38.W1 B7 /r<br>VFMSUBADD231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B          | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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[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])
    DEST[191:128] := RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192])
FI
```

#### VFMSUBADD213PD 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])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192])
FI
```

#### VFMSUBADD231PD 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])
    DEST[191:128] := RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] := RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192])
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;
        FI
      ELSE
        IF *merging-masking*           ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE                             ; zeroing-masking
          DEST[i+63:i] := 0
        FI
      FI;
    ENDFOR
  DEST[MAXVL-1:VL] := 0

```

**VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 64

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

THEN

IF j \*is even\*

THEN DEST[i+63:i] :=

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

ELSE DEST[i+63:i] :=

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j := 0 TO KL-1

i := j \* 64

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

THEN

IF j \*is even\*

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] :=

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[63:0])

ELSE

DEST[i+63:i] :=

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI;

ELSE

IF (EVEX.b = 1)

THEN

DEST[i+63:i] :=

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[63:0])

ELSE

DEST[i+63:i] :=

RoundFPControl\_MXCSR(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI;

```

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

```

```

        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
    FI;
ELSE
    IF (EVEX.b = 1)
        THEN
            DEST[i+63:i] :=
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])

            ELSE
                DEST[i+63:i] :=
                RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI;
    FI
ELSE
    IF *merging-masking*                ; merging-masking
    THEN *DEST[i+63:i] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[i+63:i] := 0
    FI
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

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

```

VFMSUBADDxxxPD __m512d __mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d __mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d __mm512_mask3_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBADDxxxPD __m256d __mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d __mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm_mask_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d __mm_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d __mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
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.”

## VFMSUBADD132PH/VFMSUBADD213PH/VFMSUBADD231PH—Fused Multiply-Alternating Subtract/Add of Packed FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP6.W0 97 /r<br>VFMSUBADD132PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and xmm3/m128/m16bcst, subtract/add elements in xmm2, and store the result in xmm1 subject to writemask k1. |
| EVEX.256.66.MAP6.W0 97 /r<br>VFMSUBADD132PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and ymm3/m256/m16bcst, subtract/add elements in ymm2, and store the result in ymm1 subject to writemask k1. |
| EVEX.512.66.MAP6.W0 97 /r<br>VFMSUBADD132PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and zmm3/m512/m16bcst, subtract/add elements in zmm2, and store the result in zmm1 subject to writemask k1. |
| EVEX.128.66.MAP6.W0 A7 /r<br>VFMSUBADD213PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1. |
| EVEX.256.66.MAP6.W0 A7 /r<br>VFMSUBADD213PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1. |
| EVEX.512.66.MAP6.W0 A7 /r<br>VFMSUBADD213PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1. |
| EVEX.128.66.MAP6.W0 B7 /r<br>VFMSUBADD231PH xmm1{k1}{z},<br>xmm2, xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm2 and xmm3/m128/m16bcst, subtract/add elements in xmm1, and store the result in xmm1 subject to writemask k1. |
| EVEX.256.66.MAP6.W0 B7 /r<br>VFMSUBADD231PH ymm1{k1}{z},<br>ymm2, ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm2 and ymm3/m256/m16bcst, subtract/add elements in ymm1, and store the result in ymm1 subject to writemask k1. |
| EVEX.512.66.MAP6.W0 B7 /r<br>VFMSUBADD231PH zmm1{k1}{z},<br>zmm2, zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values from zmm2 and zmm3/m512/m16bcst, subtract/add elements in zmm1, and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1        | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|------------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (r, w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a packed multiply-add (even elements) or multiply-subtract (odd elements) computation on FP16 values using three source operands and writes the results in the destination operand. The destination operand is also the first source operand. The notation “132”, “213” and “231” indicate the use of the operands in  $A * B \pm C$ , where each digit corresponds to the operand number, with the destination being operand 1; see Table 5-8. The destination elements are updated according to the writemask.



**Table 5-8. VFMSUBADD[132,213,231]PH Notation for Odd and Even Elements**

| Notation | Odd Elements                | Even Elements               |
|----------|-----------------------------|-----------------------------|
| 132      | $dest = dest * src3 - src2$ | $dest = dest * src3 + src2$ |
| 231      | $dest = src2 * src3 - dest$ | $dest = src2 * src3 + dest$ |
| 213      | $dest = src2 * dest - src3$ | $dest = src2 * dest + src3$ |

**Operation****VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j]\*SRC3.fp16[j] + SRC2.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j]\*SRC3.fp16[j] - SRC2.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VFMSUBADD132PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j] \* t3 + SRC2.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(DEST.fp16[j] \* t3 - SRC2.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

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

**VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*DEST.fp16[j] + SRC3.fp16[j])

ELSE

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*DEST.fp16[j] - SRC3.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VFMSUBADD213PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* DEST.fp16[j] + t3 )

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* DEST.fp16[j] - t3 )

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

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

**VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a register**

VL = 128, 256 or 512

KL := VL/16

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

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

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

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*SRC3.fp16[j] + DEST.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j]\*SRC3.fp16[j] - DEST.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VFMSUBADD231PH DEST, SRC2, SRC3 (EVEX encoded versions) when src3 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

t3 := SRC3.fp16[0]

ELSE:

t3 := SRC3.fp16[j]

IF \*j is even\*:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* t3 + DEST.fp16[j])

ELSE:

DEST.fp16[j] := RoundFPControl(SRC2.fp16[j] \* t3 - DEST.fp16[j])

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VFMSUBADD132PH, VFMSUBADD213PH, and VFMSUBADD231PH:

```

__m128h _mm_fmsubadd_ph (__m128h a, __m128h b, __m128h c);
__m128h _mm_mask_fmsubadd_ph (__m128h a, __mmask8 k, __m128h b, __m128h c);
__m128h _mm_mask3_fmsubadd_ph (__m128h a, __m128h b, __m128h c, __mmask8 k);
__m128h _mm_maskz_fmsubadd_ph (__mmask8 k, __m128h a, __m128h b, __m128h c);
__m256h _mm256_fmsubadd_ph (__m256h a, __m256h b, __m256h c);
__m256h _mm256_mask_fmsubadd_ph (__m256h a, __mmask16 k, __m256h b, __m256h c);
__m256h _mm256_mask3_fmsubadd_ph (__m256h a, __m256h b, __m256h c, __mmask16 k);
__m256h _mm256_maskz_fmsubadd_ph (__mmask16 k, __m256h a, __m256h b, __m256h c);
__m512h _mm512_fmsubadd_ph (__m512h a, __m512h b, __m512h c);
__m512h _mm512_mask_fmsubadd_ph (__m512h a, __mmask32 k, __m512h b, __m512h c);
__m512h _mm512_mask3_fmsubadd_ph (__m512h a, __m512h b, __m512h c, __mmask32 k);
__m512h _mm512_maskz_fmsubadd_ph (__mmask32 k, __m512h a, __m512h b, __m512h c);
__m512h _mm512_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask_fmsubadd_round_ph (__m512h a, __mmask32 k, __m512h b, __m512h c, const int rounding);
__m512h _mm512_mask3_fmsubadd_round_ph (__m512h a, __m512h b, __m512h c, __mmask32 k, const int rounding);
__m512h _mm512_maskz_fmsubadd_round_ph (__mmask32 k, __m512h a, __m512h b, __m512h c, const int rounding);

```

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

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

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 97 /r<br>VFMSUBADD132PS xmm1, xmm2,<br>xmm3/m128                  | A          | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.                                  |
| VEX.128.66.0F38.W0 A7 /r<br>VFMSUBADD213PS xmm1, xmm2,<br>xmm3/m128                  | A          | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.                                  |
| VEX.128.66.0F38.W0 B7 /r<br>VFMSUBADD231PS xmm1, xmm2,<br>xmm3/m128                  | A          | V/V                          | FMA                      | Multiply packed single precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.                                  |
| VEX.256.66.0F38.W0 97 /r<br>VFMSUBADD132PS ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.                                  |
| VEX.256.66.0F38.W0 A7 /r<br>VFMSUBADD213PS ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.                                  |
| VEX.256.66.0F38.W0 B7 /r<br>VFMSUBADD231PS ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | FMA                      | Multiply packed single precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.                                  |
| EVEX.128.66.0F38.W0 97 /r<br>VFMSUBADD132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W0 A7 /r<br>VFMSUBADD213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W0 B7 /r<br>VFMSUBADD231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W0 97 /r<br>VFMSUBADD132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W0 A7 /r<br>VFMSUBADD213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W0 B7 /r<br>VFMSUBADD231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| EVEX.512.66.0F38.W0 97 /r<br>VFMSUBADD132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B          | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1. |
| EVEX.512.66.0F38.W0 A7 /r<br>VFMSUBADD213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B          | V/V                          | AVX512F                  | Multiply packed single precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1. |
| EVEX.512.66.0F38.W0 B7 /r<br>VFMSUBADD231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B          | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

**VFMSUBADD132PS:** Multiplies the four, eight or sixteen packed single precision floating-point values from the first source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the second source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

**VFMSUBADD213PS:** Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the third source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

**VFMSUBADD231PS:** Multiplies the four, eight or sixteen packed single precision floating-point values from the second source operand to the corresponding packed single precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single precision floating-point elements and adds the even single precision floating-point values in the first source operand, performs rounding and stores the resulting packed single precision floating-point values to the destination operand (first source operand).

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

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

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

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

**Operation**

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

**VFMSUBADD132PS 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])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -SRC2[n+63:n+32])
}
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])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -SRC3[n+63:n+32])
}
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])
    DEST[n+63:n+32] := RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -DEST[n+63:n+32])
}
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;
    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 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
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])
                    FI
                FI
            FI
        FI
FI;

```

```

        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;
        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[j+31:i] :=
                RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] + DEST[j+31:i])
            ELSE
              DEST[j+31:i] :=
                RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] + DEST[j+31:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[j+31:i] :=
                RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[31:0] - DEST[j+31:i])
            ELSE
              DEST[j+31:i] :=
                RoundFPControl_MXCSR(SRC2[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
          FI;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[j+31:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[j+31:i] := 0
        FI
      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 __m512 __mm512_mask3_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, 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 __m_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 __m256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

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

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

## VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Negative Multiply-Add of Packed Double Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| VEX.128.66.0F38.W1 9C /r<br>VFMADD132PD xmm1, xmm2,<br>xmm3/m128                  | A         | V/V                          | FMA                      | 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.          |
| VEX.128.66.0F38.W1 AC /r<br>VFMADD213PD xmm1, xmm2,<br>xmm3/m128                  | A         | V/V                          | FMA                      | 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.          |
| VEX.128.66.0F38.W1 BC /r<br>VFMADD231PD xmm1, xmm2,<br>xmm3/m128                  | A         | V/V                          | FMA                      | 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.          |
| VEX.256.66.0F38.W1 9C /r<br>VFMADD132PD ymm1, ymm2,<br>ymm3/m256                  | A         | V/V                          | FMA                      | 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.          |
| VEX.256.66.0F38.W1 AC /r<br>VFMADD213PD ymm1, ymm2,<br>ymm3/m256                  | A         | V/V                          | FMA                      | 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.          |
| VEX.256.66.0F38.W1 BC /r<br>VFMADD231PD ymm1, ymm2,<br>ymm3/m256                  | A         | V/V                          | FMA                      | 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.          |
| EVEX.128.66.0F38.W1 9C /r<br>VFMADD132PD xmm0 {k1}{z},<br>xmm1, xmm2/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W1 AC /r<br>VFMADD213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W1 BC /r<br>VFMADD231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W1 9C /r<br>VFMADD132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W1 AC /r<br>VFMADD213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W1 BC /r<br>VFMADD231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.512.66.0F38.W1 9C /r<br>VFNMADD132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | 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. |
| EVEX.512.66.0F38.W1 AC /r<br>VFNMADD213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | 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. |
| EVEX.512.66.0F38.W1 BC /r<br>VFNMADD231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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).

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

```

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

For i = 0 to MAXNUM-1 {
    n := 64*i;
    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

```

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

```

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

For i = 0 to MAXNUM-1 {
    n := 64*i;
    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

```

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

```

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

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

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

```

**VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 64

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

THEN DEST[i+63:i] :=

RoundFPControl(-(DEST[i+63:i]\*SRC3[i+63:i]) + SRC2[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

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

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

FOR j := 0 TO KL-1

i := j \* 64

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

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] :=

RoundFPControl\_MXCSR(-(DEST[i+63:i]\*SRC3[63:0]) + SRC2[i+63:i])

ELSE

DEST[i+63:i] :=

RoundFPControl\_MXCSR(-(DEST[i+63:i]\*SRC3[j+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 register)**

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 64

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

THEN DEST[i+63:i] :=

RoundFPControl(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j := 0 TO KL-1

i := j \* 64

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

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] :=

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[63:0])

ELSE

DEST[i+63:i] :=

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

IF (VL = 512) AND (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[63:0]) + DEST[i+63:i])

ELSE

DEST[i+63:i] :=

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*SRC3[j+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_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMADDxxxPD __m256d __mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d __mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d __mm_mask_fnmadd_pd(__m128d a, __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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| VEX.128.66.0F38.W0 9C /r<br>VFNMADD132PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | 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.          |
| VEX.128.66.0F38.W0 AC /r<br>VFNMADD213PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | 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.          |
| VEX.128.66.0F38.W0 BC /r<br>VFNMADD231PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | 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.          |
| VEX.256.66.0F38.W0 9C /r<br>VFNMADD132PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | 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.          |
| VEX.256.66.0F38.W0 AC /r<br>VFNMADD213PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | 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.          |
| VEX.256.66.0F38.0 BC /r<br>VFNMADD231PS ymm1, ymm2,<br>ymm3/m256                       | A         | V/V                          | FMA                      | 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.          |
| EVEX.128.66.0F38.W0 9C /r<br>VFNMADD132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W0 AC /r<br>VFNMADD213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W0 BC /r<br>VFNMADD231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W0 9C /r<br>VFNMADD132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W0 AC /r<br>VFNMADD213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W0 BC /r<br>VFNMADD231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F38.W0 9C /r<br>VFNMADD132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F38.W0 AC /r<br>VFNMADD213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | 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. |
| EVEX.512.66.0F38.W0 BC /r<br>VFNMADD231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | 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. |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## Description

**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 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).

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

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

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

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

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

## Operation

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

**VFMADD132PS 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

```

**VFMADD213PS 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

```

**VFMADD231PS 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

```

**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_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 _mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_maskz_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask3_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 _mm256_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_maskz_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask3_fnmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_mask3_fnmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

VEX-encoded instructions, see 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

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| VEX.LIG.66.0F38.W1 9D /r<br>VFNMADD132SD xmm1, xmm2,<br>xmm3/m64               | A          | V/V                          | FMA                      | 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. |
| VEX.LIG.66.0F38.W1 AD /r<br>VFNMADD213SD xmm1, xmm2,<br>xmm3/m64               | A          | V/V                          | FMA                      | 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. |
| VEX.LIG.66.0F38.W1 BD /r<br>VFNMADD231SD xmm1, xmm2,<br>xmm3/m64               | A          | V/V                          | FMA                      | 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. |
| EVEX.LLIG.66.0F38.W1 9D /r<br>VFNMADD132SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | B          | V/V                          | AVX512F                  | 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. |
| EVEX.LLIG.66.0F38.W1 AD /r<br>VFNMADD213SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | B          | V/V                          | AVX512F                  | 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. |
| EVEX.LLIG.66.0F38.W1 BD /r<br>VFNMADD231SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | B          | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | N/A           | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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).

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

```

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

```

VFMADDxxxSD __m128d __mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d __mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d __mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d __mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d __mm_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 Negative Multiply-Add of Scalar Single Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| VEX.LIG.66.0F38.W0 9D /r<br>VFMADD132SS xmm1, xmm2,<br>xmm3/m32               | A          | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1. |
| VEX.LIG.66.0F38.W0 AD /r<br>VFMADD213SS xmm1, xmm2,<br>xmm3/m32               | A          | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1. |
| VEX.LIG.66.0F38.W0 BD /r<br>VFMADD231SS xmm1, xmm2,<br>xmm3/m32               | A          | V/V                          | FMA                      | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W0 9D /r<br>VFMADD132SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | B          | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W0 AD /r<br>VFMADD213SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | B          | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W0 BD /r<br>VFMADD231SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | B          | V/V                          | AVX512F                  | Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | N/A           | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

**VFMADD132SS:** 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).

**VFMADD213SS:** 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).

**VFMADD231SS:** 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).

#### 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[127:32] := DEST[127:32]
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[127:32] := DEST[127:32]
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[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[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

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

```

DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

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

```

DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

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

```

DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

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

```

VFMADDxxxSS __m128 __mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 __mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 __mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 __mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 __mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 __mm_fmadd_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.”

## VFNSUB132PD/VFNSUB213PD/VFNSUB231PD—Fused Negative Multiply-Subtract of Packed Double Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| VEX.128.66.0F38.W1 9E /r<br>VFNSUB132PD xmm1, xmm2,<br>xmm3/m128                  | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.          |
| VEX.128.66.0F38.W1 AE /r<br>VFNSUB213PD xmm1, xmm2,<br>xmm3/m128                  | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.          |
| VEX.128.66.0F38.W1 BE /r<br>VFNSUB231PD xmm1, xmm2,<br>xmm3/m128                  | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.          |
| VEX.256.66.0F38.W1 9E /r<br>VFNSUB132PD ymm1, ymm2,<br>ymm3/m256                  | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.          |
| VEX.256.66.0F38.W1 AE /r<br>VFNSUB213PD ymm1, ymm2,<br>ymm3/m256                  | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.          |
| VEX.256.66.0F38.W1 BE /r<br>VFNSUB231PD ymm1, ymm2,<br>ymm3/m256                  | A         | V/V                          | FMA                      | Multiply packed double precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.          |
| EVEX.128.66.0F38.W1 9E /r<br>VFNSUB132PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W1 AE /r<br>VFNSUB213PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.128.66.0F38.W1 BE /r<br>VFNSUB231PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W1 9E /r<br>VFNSUB132PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W1 AE /r<br>VFNSUB213PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W1 BE /r<br>VFNSUB231PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |



| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.512.66.0F38.W1 9E /r<br>VFNMSUB132PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | 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. |
| EVEX.512.66.0F38.W1 AE /r<br>VFNMSUB213PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | 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. |
| EVEX.512.66.0F38.W1 BE /r<br>VFNMSUB231PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst{er} | B         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

#### 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 YMM register or a 256-bit memory location and encoded in `rm_field`.

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

#### Operation

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

**VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)**

```

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

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

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

```

**VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)**

```

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

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

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

```

**VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)**

```

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

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

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

```

**VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 64

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

THEN DEST[i+63:i] :=

RoundFPControl(-(DEST[i+63:i]\*SRC3[i+63:i]) - SRC2[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j := 0 TO KL-1

i := j \* 64

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

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] :=

RoundFPControl\_MXCSR(-(DEST[i+63:i]\*SRC3[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

**VFNSUB213PD 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

**VFNSUB213PD 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

**VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

FOR j := 0 TO KL-1

i := j \* 64

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

THEN DEST[i+63:i] :=

RoundFPControl(-(SRC2[i+63:i]\*SRC3[i+63:i]) - DEST[i+63:i])

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j := 0 TO KL-1

i := j \* 64

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

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] :=

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*SRC3[63:0]) - DEST[i+63:i])

ELSE

DEST[i+63:i] :=

RoundFPControl\_MXCSR(-(SRC2[i+63:i]\*SRC3[i+63:i]) - DEST[i+63:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

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

```

VFNSUBxxxPD __m512d _mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNSUBxxxPD __m512d _mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNSUBxxxPD __m512d _mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNSUBxxxPD __m512d _mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNSUBxxxPD __m512d _mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNSUBxxxPD __m512d _mm512_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNSUBxxxPD __m512d _mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNSUBxxxPD __m512d _mm512_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNSUBxxxPD __m256d _mm256_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNSUBxxxPD __m256d _mm256_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNSUBxxxPD __m256d _mm256_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNSUBxxxPD __m128d _mm_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNSUBxxxPD __m128d _mm_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNSUBxxxPD __m128d _mm_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNSUBxxxPD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNSUBxxxPD __m256d _mm256_fnmsub_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.”

## VFNSUB132PS/VFNSUB213PS/VFNSUB231PS—Fused Negative Multiply-Subtract of Packed Single Precision Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 9E /r<br>VFNSUB132PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.             |
| VEX.128.66.0F38.W0 AE /r<br>VFNSUB213PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.             |
| VEX.128.66.0F38.W0 BE /r<br>VFNSUB231PS xmm1, xmm2,<br>xmm3/m128                      | A         | V/V                          | FMA                      | Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.             |
| VEX.256.66.0F38.W0 9E /r<br>VFNSUB132PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.             |
| VEX.256.66.0F38.W0 AE /r<br>VFNSUB213PS ymm1, ymm2,<br>ymm3/m256                      | A         | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.             |
| VEX.256.66.0F38.0 BE /r<br>VFNSUB231PS ymm1, ymm2,<br>ymm3/m256                       | A         | V/V                          | FMA                      | Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.             |
| EVEX.128.66.0F38.W0 9E /r<br>VFNSUB132PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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.    |
| EVEX.128.66.0F38.W0 AE /r<br>VFNSUB213PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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.    |
| EVEX.128.66.0F38.W0 BE /r<br>VFNSUB231PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W0 9E /r<br>VFNSUB132PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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.    |
| EVEX.256.66.0F38.W0 AE /r<br>VFNSUB213PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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.    |
| EVEX.256.66.0F38.W0 BE /r<br>VFNSUB231PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst     | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F38.W0 9E /r<br>VFNSUB132PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | 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.    |
| EVEX.512.66.0F38.W0 AE /r<br>VFNSUB213PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | 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.    |
| EVEX.512.66.0F38.W0 BE /r<br>VFNSUB231PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst{er} | B         | V/V                          | AVX512F                  | 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. |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## Description

**VFNSUB132PS:** 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).

**VFNSUB213PS:** 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).

**VFNSUB231PS:** 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).

**VFNSUB132PS 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

```

**VFNSUB132PS 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[j+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

```

**VFNSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

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

```

**VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

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

```

**VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b = 1)
        THEN
          DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
        ELSE
          DEST[i+31:i] :=
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[j+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 __m512 __mm512_mask3_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMSUBxxxPS __m256 __mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 __mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 __mm_mask_fnmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __mm_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 __mm_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m128 __mm_fnmsub_ps (__m128 a, __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.”

## VFNSUB132SD/VFNSUB213SD/VFNSUB231SD—Fused Negative Multiply-Subtract of Scalar Double Precision Floating-Point Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| VEX.LIG.66.0F38.W1 9F /r<br>VFNSUB132SD xmm1, xmm2,<br>xmm3/m64               | A          | V/V                          | FMA                      | Multiply scalar double precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1. |
| VEX.LIG.66.0F38.W1 AF /r<br>VFNSUB213SD xmm1, xmm2,<br>xmm3/m64               | A          | V/V                          | FMA                      | Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1. |
| VEX.LIG.66.0F38.W1 BF /r<br>VFNSUB231SD xmm1, xmm2,<br>xmm3/m64               | A          | V/V                          | FMA                      | Multiply scalar double precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W1 9F /r<br>VFNSUB132SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | B          | V/V                          | AVX512F                  | Multiply scalar double precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W1 AF /r<br>VFNSUB213SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | B          | V/V                          | AVX512F                  | Multiply scalar double precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1. |
| EVEX.LLIG.66.0F38.W1 BF /r<br>VFNSUB231SD xmm1 {k1}{z},<br>xmm2, xmm3/m64{er} | B          | V/V                          | AVX512F                  | Multiply scalar double precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | N/A           | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

**VFNSUB132SD:** 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).

**VFNSUB213SD:** 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).

**VFNSUB231SD:** 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).

#### VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] := RoundFPControl(-(DEST[63:0]\*SRC3[63:0]) - SRC2[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] := 0

FI;

FI;

DEST[127:64] := DEST[127:64]

DEST[MAXVL-1:128] := 0

#### VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

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

THEN

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(EVEX.RC);

ELSE

SET\_ROUNDING\_MODE\_FOR\_THIS\_INSTRUCTION(MXCSR.RC);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] := RoundFPControl(-(SRC2[63:0]\*DEST[63:0]) - SRC3[63:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] := 0

FI;

FI;

DEST[127:64] := DEST[127:64]

DEST[MAXVL-1:128] := 0

**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**

```

VFNMSUBxxxSD __m128d __mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d __mm_mask_fnmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d __mm_maskz_fnmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d __mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxSD __m128d __mm_mask_fnmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d __mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d __mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSUBxxxSD __m128d __mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

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

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

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

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|------------|------------------------------|--------------------------|--|
| VEX.LIG.66.0F38.W0 9F /r<br>VFNMSUB132SS xmm1, xmm2,<br>xmm3/m32               | 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<br>VFNMSUB213SS xmm1, xmm2,<br>xmm3/m32               | 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<br>VFNMSUB231SS xmm1, xmm2,<br>xmm3/m32               | 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.LLIG.66.0F38.W0 9F /r<br>VFNMSUB132SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | 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.LLIG.66.0F38.W0 AF /r<br>VFNMSUB213SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | 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.LLIG.66.0F38.W0 BF /r<br>VFNMSUB231SS xmm1 {k1}{z},<br>xmm2, xmm3/m32{er} | 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

| Op/En | Tuple Type    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|------------------|---------------|---------------|-----------|
| A     | N/A           | ModRM:reg (r, w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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[127:32] := DEST[127:32]
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[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

**VFNMSSUB231SS 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[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

**VFNMSSUB132SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] := RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

**VFNMSSUB213SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

**VFNMSSUB231SS DEST, SRC2, SRC3 (VEX encoded version)**

```

DEST[31:0] := RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[127:32] := DEST[127:32]
DEST[MAXVL-1:128] := 0

```

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

```

VFNMSSUBxxxSS __m128 __mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSSUBxxxSS __m128 __mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSSUBxxxSS __m128 __mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSSUBxxxSS __m128 __mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSSUBxxxSS __m128 __mm_mask_fnmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMSSUBxxxSS __m128 __mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSSUBxxxSS __m128 __mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSSUBxxxSS __m128 __mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal.

**Other Exceptions**

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

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

## VFPCCLASSPD—Tests Types of Packed Float64 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W1 66 /r ib<br>VFPCCLASSPD k2 {k1},<br>xmm2/m128/m64bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512DQ     | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.256.66.0F3A.W1 66 /r ib<br>VFPCCLASSPD k2 {k1},<br>ymm2/m256/m64bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512DQ     | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.512.66.0F3A.W1 66 /r ib<br>VFPCCLASSPD k2 {k1},<br>zmm2/m512/m64bcst, imm8 | A         | V/V                          | AVX512DQ                 | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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.

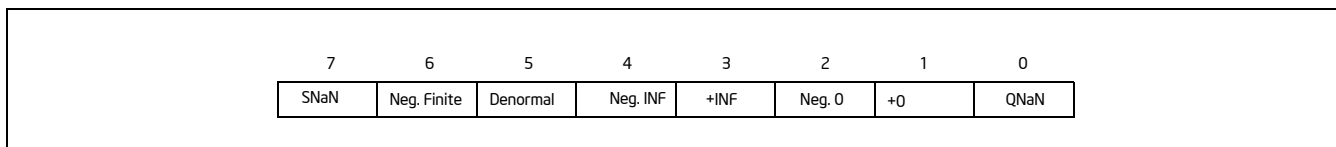


Figure 5-13. Imm8 Byte Specifier of Special Case Floating-Point Values for VFPCCLASSPD/SD/PS/SS

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

| Bits       | Imm8[0]         | Imm8[1]       | Imm8[2]       | Imm8[3]         | Imm8[4]         | Imm8[5]             | Imm8[6]                    | Imm8[7]         |
|------------|-----------------|---------------|---------------|-----------------|-----------------|---------------------|----------------------------|-----------------|
| Category   | QNaN            | PosZero       | NegZero       | PosINF          | NegINF          | Denormal            | Negative                   | SNaN            |
| Classifier | Checks for QNaN | Checks for +0 | Checks for -0 | Checks for +INF | Checks for -INF | Checks for Denormal | Checks for Negative finite | Checks for SNaN |

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

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

### Operation

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

```

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

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

    bResult = ( imm8[0] AND qNaN_res ) OR ( imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} /* end of CheckFPClassDP() */

```

### VFPCLASSPD (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 __mm_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.

## VFPCLASSPH—Test Types of Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.NP.0F3A.W0 66 /r /ib<br>VFPCLASSPH k1{k2}, xmm1/m128/<br>m16bcst, imm8 | A         | V/V                          | AVX512-FP16<br>AVX512VL | Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.256.NP.0F3A.W0 66 /r /ib<br>VFPCLASSPH k1{k2}, ymm1/m256/<br>m16bcst, imm8 | A         | V/V                          | AVX512-FP16<br>AVX512VL | Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.512.NP.0F3A.W0 66 /r /ib<br>VFPCLASSPH k1{k2}, zmm1/m512/<br>m16bcst, imm8 | A         | V/V                          | AVX512-FP16             | Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | imm8 (r)  | N/A       |

### Description

This instruction checks the packed FP16 values 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; see Table 5-9 for the categories. 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 corresponding bits in the destination mask register according to the writemask.

**Table 5-9. Classifier Operations for VFPCLASSPH/VFPCLASSSH**

| Bits    | Category | Classifier                 |
|---------|----------|----------------------------|
| imm8[0] | QNaN     | Checks for QNaN            |
| imm8[1] | PosZero  | Checks +0                  |
| imm8[2] | NegZero  | Checks for -0              |
| imm8[3] | PosINF   | Checks for $+\infty$       |
| imm8[4] | NegINF   | Checks for $-\infty$       |
| imm8[5] | Denormal | Checks for Denormal        |
| imm8[6] | Negative | Checks for Negative finite |
| imm8[7] | SNAN     | Checks for SNAN            |

**Operation**

```

def check_fp_class_fp16(tsrc, imm8):
    negative := tsrc[15]
    exponent_all_ones := (tsrc[14:10] == 0x1F)
    exponent_all_zeros := (tsrc[14:10] == 0)
    mantissa_all_zeros := (tsrc[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := tsrc[9]

    snan := exponent_all_ones and not(mantissa_all_zeros) and not(signaling_bit)
    qnan := exponent_all_ones and not(mantissa_all_zeros) and signaling_bit
    positive_zero := not(negative) and zero
    negative_zero := negative and zero
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    negative_infinity := negative and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)
    finite_negative := negative and not(exponent_all_ones) and not(zero)

    return (imm8[0] and qnan) OR
           (imm8[1] and positive_zero) OR
           (imm8[2] and negative_zero) OR
           (imm8[3] and positive_infinity) OR
           (imm8[4] and negative_infinity) OR
           (imm8[5] and denormal) OR
           (imm8[6] and finite_negative) OR
           (imm8[7] and snan)

```

**VFPCLASSPH dest{k2}, src, imm8**

VL = 128, 256 or 512

KL := VL/16

```

FOR i := 0 to KL-1:
    IF k2[i] or *no writemask*:
        IF SRC is memory and (EVEX.b = 1):
            tsrc := SRC.fp16[0]
        ELSE:
            tsrc := SRC.fp16[i]
        DEST.bit[i] := check_fp_class_fp16(tsrc, imm8)
    ELSE:
        DEST.bit[i] := 0

```

DEST[MAXKL-1:kl] := 0

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

```

VFPCLASSPH __mmask8 _mm_fpclass_ph_mask (__m128h a, int imm8);
VFPCLASSPH __mmask8 _mm_mask_fpclass_ph_mask (__mmask8 k1, __m128h a, int imm8);
VFPCLASSPH __mmask16 _mm256_fpclass_ph_mask (__m256h a, int imm8);
VFPCLASSPH __mmask16 _mm256_mask_fpclass_ph_mask (__mmask16 k1, __m256h a, int imm8);
VFPCLASSPH __mmask32 _mm512_fpclass_ph_mask (__m512h a, int imm8);
VFPCLASSPH __mmask32 _mm512_mask_fpclass_ph_mask (__mmask32 k1, __m512h a, int imm8);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

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



## VFPCCLASSPS—Tests Types of Packed Float32 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W0 66 /r ib<br>VFPCCLASSPS k2 {k1},<br>xmm2/m128/m32bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512DQ     | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.256.66.0F3A.W0 66 /r ib<br>VFPCCLASSPS k2 {k1},<br>ymm2/m256/m32bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512DQ     | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |
| EVEX.512.66.0F3A.W0 66 /r ib<br>VFPCCLASSPS k2 {k1},<br>zmm2/m512/m32bcst, imm8 | A         | V/V                          | AVX512DQ                 | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

The FPCCLASSPS 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]=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
           ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
           ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
           ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
Return bResult;
} /* end of CheckSPClassSP() */

```

**VFPCLASSPS (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[j+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**

```

VFPCLASSPS __mmask16 __mm512_fpclass_ps_mask( __m512 a, int c);
VFPCLASSPS __mmask16 __mm512_mask_fpclass_ps_mask( __mmask16 m, __m512 a, int c)
VFPCLASSPS __mmask8 __mm256_fpclass_ps_mask( __m256 a, int c)
VFPCLASSPS __mmask8 __mm256_mask_fpclass_ps_mask( __mmask8 m, __m256 a, int c)
VFPCLASSPS __mmask8 __mm_fpclass_ps_mask( __m128 a, int c)
VFPCLASSPS __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.

## VFPCCLASSSD—Tests Type of a Scalar Float64 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.LLIG.66.0F3A.W1 67 /r ib<br>VFPCCLASSSD k2 {k1},<br>xmm2/m64, imm8 | A         | V/V                          | AVX512DQ                 | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

The FPCLASSSD instruction checks the low double precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX\_KL-1: 1 of the destination are cleared.

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

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

### Operation

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

```

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

    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
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} /* end of CheckFPClassDP() */
```

**VFPClassSD (EVEX encoded version)**

```
IF k1[0] OR *no writemask*  
  THEN DEST[0] :=  
    CheckFPClassDP(SRC1[63:0], imm8[7:0])  
  ELSE DEST[0] := 0 ; zeroing-masking only  
FI;  
DEST[MAX_KL-1:1] := 0
```

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

```
VFPClassSD __mmask8 _mm_fpclass_sd_mask( __m128d a, int c)  
VFPClassSD __mmask8 _mm_mask_fpclass_sd_mask( __mmask8 m, __m128d a, int c)
```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD If EVEX.vvvv != 1111B.

## VFPCLASSSH—Test Types of Scalar FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|--|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.NP.OF3A.WO 67 /r /ib<br>VFPCLASSSH k1{k2}, xmm1/m16,<br>imm8 | A         | V/V                          | AVX512-FP16           | Test the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------|---------------|---------------|-----------|-----------|
| A     | Scalar | ModRM:reg (w) | ModRM:r/m (r) | imm8 (r)  | N/A       |

### Description

This instruction checks the low FP16 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; see Table 5-9 for the categories. 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 the destination mask register according to the writemask. The other bits in the destination mask register are zeroed.

### Operation

#### VFPCLASSSH dest{k2}, src, imm8

IF k2[0] or \*no writemask\*:

```
DEST.bit[0] := check_fp_class_fp16(src.fp16[0], imm8) // see VFPCLASSPH
```

ELSE:

```
DEST.bit[0] := 0
```

```
DEST[MAXKL-1:1] := 0
```

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

```
VFPCLASSSH __mmask8 __mm_fpclass_sh_mask (__m128h a, int imm8);
```

```
VFPCLASSSH __mmask8 __mm_mask_fpclass_sh_mask (__mmask8 k1, __m128h a, int imm8);
```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

EVEX-encoded instructions, see Table 2-58, “Type E10 Class Exception Conditions.”

## VFPCLASSSS—Tests Type of a Scalar Float32 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.LLIG.66.0F3A.W0 67 /r<br>VFPCLASSSS k2 {k1},<br>xmm2/m32, imm8 | A         | V/V                          | AVX512DQ                 | Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

The FPCLASSSS instruction checks the low single-precision floating-point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX\_KL-1: 1 of the destination are cleared.

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

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
           ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
           ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
           ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
Return bResult;

```

```
 } /* end of CheckSPClassSP() */
```

### VFPCLASSSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
  THEN DEST[0] :=
      CheckFPClassSP(SRC1[31:0], imm8[7:0])
  ELSE DEST[0] := 0          ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] := 0
```

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

```
VFPCLASSSS __mmask8 _mm_fpclass_ss_mask( __m128 a, int c)
```

```
VFPCLASSSS __mmask8 _mm_mask_fpclass_ss_mask( __mmask8 m, __m128 a, int c)
```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

```
#UD          If EVEX.vvvv != 1111B.
```

## VGATHERDPD/VGATHERQPD—Gather Packed Double Precision Floating-Point Values Using Signed Dword/Qword Indices

| Opcode/<br>Instruction                                   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| VEX.128.66.0F38.W1 92 /r<br>VGATHERDPD xmm1, vm32x, xmm2 | RMV       | V/V                          | AVX2                     | Using dword indices specified in vm32x, gather double precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |
| VEX.128.66.0F38.W1 93 /r<br>VGATHERQPD xmm1, vm64x, xmm2 | RMV       | V/V                          | AVX2                     | Using qword indices specified in vm64x, gather double precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |
| VEX.256.66.0F38.W1 92 /r<br>VGATHERDPD ymm1, vm32x, ymm2 | RMV       | V/V                          | AVX2                     | Using dword indices specified in vm32x, gather double precision floating-point values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1. |
| VEX.256.66.0F38.W1 93 /r<br>VGATHERQPD ymm1, vm64y, ymm2 | RMV       | V/V                          | AVX2                     | Using qword indices specified in vm64y, gather double precision floating-point values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1. |

### Instruction Operand Encoding

| Op/En | Operand 1       | Operand 2   | Operand 3       | Operand 4 |
|-------|-----------------|---|-----------------|-----------|
| RMV   | ModRM:reg (r,w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | VEX.vvvv (r, w) | N/A       |

### Description

The instruction conditionally loads up to 2 or 4 double precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two double precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.



VEX.256 version: The instruction will gather four double precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST := SRC1;

BASE\_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK := SRC3;

### VGATHERDPD (VEX.128 version)

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

```
FOR j := 0 to 1
```

```
  i := j * 64;
```

```
  IF MASK[63+i] THEN
```

```
    MASK[j +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
```

```
  ELSE
```

```
    MASK[j +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[j +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
```

```
  FI;
```

```
  MASK[j +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[j +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[j +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, __m256i index, const int scale);`

VGATHERQPD: `__m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Table 2-27, "Type 12 Class Exception Conditions."

## VGATHERDPS/VGATHERQPS—Gather Packed Single Precision Floating-Point Values Using Signed Dword/Qword Indices

| Opcode/<br>Instruction                                   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| VEX.128.66.0F38.W0 92 /r<br>VGATHERDPS xmm1, vm32x, xmm2 | A         | V/V                          | AVX2                     | Using dword indices specified in vm32x, gather single-precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |
| VEX.128.66.0F38.W0 93 /r<br>VGATHERQPS xmm1, vm64x, xmm2 | A         | V/V                          | AVX2                     | Using qword indices specified in vm64x, gather single-precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |
| VEX.256.66.0F38.W0 92 /r<br>VGATHERDPS ymm1, vm32y, ymm2 | A         | V/V                          | AVX2                     | Using dword indices specified in vm32y, gather single-precision floating-point values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1. |
| VEX.256.66.0F38.W0 93 /r<br>VGATHERQPS xmm1, vm64y, xmm2 | A         | V/V                          | AVX2                     | Using qword indices specified in vm64y, gather single-precision floating-point values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1       | Operand 2   | Operand 3       | Operand 4 |
|-------|-----------------|---|-----------------|-----------|
| A     | ModRM:reg (r,w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | VEX.vvvv (r, w) | N/A       |

### Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zero the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zero the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST := SRC1;  
 BASE\_ADDR: base register encoded in VSIB addressing;  
 VINDEX: the vector index register encoded by VSIB addressing;  
 SCALE: scale factor encoded by SIB:[7:6];  
 DISP: optional 1, 4 byte displacement;  
 MASK := SRC3;

### VGATHERDPS (VEX.128 version)

```

MASK[MAXVL-1:128] := 0;
FOR j := 0 to 3
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 3
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i + 31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] := 0;
ENDFOR

```

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

**VGATHERQPS (VEX.128 version)**

```
MASK[MAXVL-1:64] := 0;
FOR j := 0 to 3
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 1
  k := j * 64;
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:64] := 0;
```

**VGATHERDPS (VEX.256 version)**

```
MASK[MAXVL-1:256] := 0;
FOR j := 0 to 7
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 7
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;
```

**VGATHERQPS (VEX.256 version)**

```
MASK[MAXVL-1:128] := 0;
FOR j := 0 to 7
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 3
```

```

k := j * 64;
i := j * 32;
DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
IF MASK[31:i] THEN
    DEST[j +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
FI;
MASK[j +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:  __m128 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);
VGATHERQPS:  __m128 _mm256_mask_i64gather_ps (__m128 src, float const * base, __m256i index, __m128 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 Indices

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EEX.128.66.0F38.W0 92 /vsib<br>VGATHERDPS xmm1 {k1}, vm32x    | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask. |
| EEX.256.66.0F38.W0 92 /vsib<br>VGATHERDPS ymm1 {k1}, vm32y    | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask. |
| EEX.512.66.0F38.W0 92 /vsib<br>VGATHERDPS zmm1 {k1}, vm32z    | A         | V/V                          | AVX512F                  | Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask. |
| EEX.128.66.0F38.W1 92 /vsib<br>VGATHERDPD xmm1 {k1},<br>vm32x | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.            |
| EEX.256.66.0F38.W1 92 /vsib<br>VGATHERDPD ymm1 {k1},<br>vm32x | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.            |
| EEX.512.66.0F38.W1 92 /vsib<br>VGATHERDPD zmm1 {k1}, vm32y    | A         | V/V                          | AVX512F                  | Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.            |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2   | Operand 3 | Operand 4 |
|-------|---------------|---------------|---|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | N/A       | N/A       |

### Description

A set of single-precision/double precision floating-point memory locations pointed by base address `BASE_ADDR` and index vector `V_INDEX` with scale `SCALE` are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this



instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  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_mask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d __mm_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_mask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128 __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

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

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2   | Operand 3 | Operand 4 |
|-------|---------------|---------------|---|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | N/A       | N/A       |

### Description

A set of 8 single-precision/double precision floating-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  $\text{disp8} * N$  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 * 32
    k := j * 64
    IF  $k1[j]$  OR *no writemask*
        THEN  $DEST[i+31:i] :=$ 
             $MEM[BASE\_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]$ 
             $k1[j] := 0$ 
        ELSE * $DEST[i+31:i] :=$  remains unchanged*
    FI;
ENDFOR
 $k1[MAX\_KL-1:KL] := 0$ 
 $DEST[MAXVL-1:VL/2] := 0$ 

```

### VGATHERQPD (EVEX encoded version)

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

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

```

    i := j * 64
    IF  $k1[j]$  OR *no writemask*
        THEN  $DEST[i+63:i] := MEM[BASE\_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]$ 
             $k1[j] := 0$ 
        ELSE * $DEST[i+63:i] :=$  remains unchanged*
    FI;
ENDFOR
 $k1[MAX\_KL-1:KL] := 0$ 
 $DEST[MAXVL-1:VL] := 0$ 

```

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

```

VGATHERQPD __m512d __mm512_i64gather_pd(__m512i vdx, void * base, int scale);
VGATHERQPD __m512d __mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d __mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d __mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_i64gather_ps(__m512i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 __mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 __mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-61, “Type E12 Class Exception Conditions.”

## VGETEXPPD—Convert Exponents of Packed Double Precision Floating-Point Values to Double Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W1 42 /r<br>VGETEXPPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst      | A         | V/V                          | AVX512VL<br>AVX512F      | Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.           |
| EVEX.256.66.0F38.W1 42 /r<br>VGETEXPPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst      | A         | V/V                          | AVX512VL<br>AVX512F      | Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination register.           |
| EVEX.512.66.0F38.W1 42 /r<br>VGETEXPPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae} | A         | V/V                          | AVX512F                  | Convert the exponent of packed double precision floating-point values in the source operand to double precision floating-point results representing unbiased integer exponents and stores the results in the destination under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

Extracts the biased exponents from the normalized double precision floating-point 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 floating-point value and written to the corresponding qword elements of the destination operand (the first operand) as double precision floating-point 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 floating-point 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 **floor(x)** stands for the greatest integer not exceeding real number x.

**Table 5-5. VGETEXPPD/SD Special Cases**

| Input Operand                    | Result                                | Comments   |
|----------------------------------|---------------------------------------|--|
| src1 = NaN                       | QNaN(src1)                            | If (SRC = SNaN) then #IE<br>If (SRC = denormal) then #DE |
| $0 <  \text{src1}  < \text{INF}$ | $\text{floor}(\log_2( \text{src1} ))$ |  |
| $ \text{src1}  = +\text{INF}$    | +INF                                  |  |
| $ \text{src1}  = 0$              | -INF                                  |  |

**Operation**

```
NormalizeExpTinyDPFP(SRC[63:0])
```

```
{
    // Jbit is the hidden integral bit of a floating-point number. In case of denormal number it has the value of ZERO.
    Src.Jbit := 0;
    Dst.exp := 1;
    Dst.fraction := SRC[51:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit := Dst.fraction[51];          // Get the fraction MSB
        Dst.fraction := Dst.fraction << 1;    // One bit shift left
        Dst.exp--;                            // Decrement the exponent
    }
    Dst.fraction := 0;                        // zero out fraction bits
    Dst.sign := 1;                            // Return negative sign
    TMP[63:0] := MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
    Return (TMP[63:0]);
}
```

```
ConvertExpDPFP(SRC[63:0])
```

```
{
    Src.sign := 0;                            // Zero out sign bit
    Src.exp := SRC[62:52];
    Src.fraction := SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (Src = +INF) RETURN (Src);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
}
ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
{
    IF ((Src.exp = 0) AND (Src.fraction != 0))
    {
        TMP[63:0] := NormalizeExpTinyDPFP(SRC[63:0]); // Get Normalized Exponent
        Set #DE
    }
    ELSE // exponent value is correct
    {
        TMP[63:0] := (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction);
    }
    TMP := SAR(TMP, 52); // Shift Arithmetic Right
    TMP := TMP - 1023; // Subtract Bias
    Return CvtI2D(TMP); // Convert INT to double precision floating-point 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 != 1111B.



## VGETEXPPH—Convert Exponents of Packed FP16 Values to FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVE <sub>X</sub> .128.66.MAP6.WO 42 /r<br>VGETEXPPH xmm1{k1}{z}, xmm2/<br>m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1. |
| EVE <sub>X</sub> .256.66.MAP6.WO 42 /r<br>VGETEXPPH ymm1{k1}{z}, ymm2/<br>m256/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1. |
| EVE <sub>X</sub> .512.66.MAP6.WO 42 /r<br>VGETEXPPH zmm1{k1}{z}, zmm2/<br>m512/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Convert the exponent of FP16 values in the source operand to FP16 results representing unbiased integer exponents and stores the results in the destination register subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction extracts the biased exponents from the normalized FP16 representation of each word 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 an FP16 value and written to the corresponding word elements of the destination operand (the first operand) as FP16 numbers.

The destination elements are updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-6.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPH). Thus, the VGETEXPPH instruction does not require software to handle SIMD floating-point exceptions.

**Table 5-6. VGETEXPPH/VGETEXPSH Special Cases**

| Input Operand    | Result                           | Comments   |
|------------------|----------------------------------|--|
| src1 = NaN       | QNaN(src1)                       | If (SRC = SNaN), then #IE.<br>If (SRC = denormal), then #DE. |
| 0 <  src1  < INF | floor(log <sub>2</sub> ( src1 )) |  |
| src1  = +INF     | +INF                             |  |
| src1  = 0        | -INF                             |  |

**Operation**

```

def normalize_exponent_tiny_fp16(src):
    jbit := 0
    // src & dst are FP16 numbers with sign(1b), exp(5b) and fraction (10b) fields
    dst.exp := 1 // write bits 14:10
    dst.fraction := src.fraction // copy bits 9:0
    while jbit == 0:
        jbit := dst.fraction[9] // msb of the fraction
        dst.fraction := dst.fraction << 1
        dst.exp := dst.exp - 1
    dst.fraction := 0
    return dst

def getexp_fp16(src):
    src.sign := 0 // make positive
    exponent_all_ones := (src[14:10] == 0x1F)
    exponent_all_zeros := (src[14:10] == 0)
    mantissa_all_zeros := (src[9:0] == 0)
    zero := exponent_all_zeros and mantissa_all_zeros
    signaling_bit := src[9]

    nan := exponent_all_ones and not(mantissa_all_zeros)
    snan := nan and not(signaling_bit)
    qnan := nan and signaling_bit
    positive_infinity := not(negative) and exponent_all_ones and mantissa_all_zeros
    denormal := exponent_all_zeros and not(mantissa_all_zeros)

    if nan:
        if snan:
            MXCSR.IE := 1
            return qnan(src) // convert snan to a qnan
    if positive_infinity:
        return src
    if zero:
        return -INF
    if denormal:
        tmp := normalize_exponent_tiny_fp16(src)
        MXCSR.DE := 1
    else:
        tmp := src
    tmp := SAR(tmp, 10) // shift arithmetic right
    tmp := tmp - 15 // subtract bias
    return convert_integer_to_fp16(tmp)

```

**VGETEXPPH dest{k1}, src**

VL = 128, 256 or 512

KL := VL/16

FOR i := 0 to KL-1:

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

IF SRC is memory and (EVEX.b = 1):

tsrc := src.fp16[0]

ELSE:

tsrc := src.fp16[i]

DEST.fp16[i] := getexp\_fp16(tsrc)

ELSE IF \*zeroing\*:

DEST.fp16[i] := 0

//else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VGETEXPPH \_\_m128h \_mm\_getexp\_ph (\_\_m128h a);

VGETEXPPH \_\_m128h \_mm\_mask\_getexp\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a);

VGETEXPPH \_\_m128h \_mm\_maskz\_getexp\_ph (\_\_mmask8 k, \_\_m128h a);

VGETEXPPH \_\_m256h \_mm256\_getexp\_ph (\_\_m256h a);

VGETEXPPH \_\_m256h \_mm256\_mask\_getexp\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a);

VGETEXPPH \_\_m256h \_mm256\_maskz\_getexp\_ph (\_\_mmask16 k, \_\_m256h a);

VGETEXPPH \_\_m512h \_mm512\_getexp\_ph (\_\_m512h a);

VGETEXPPH \_\_m512h \_mm512\_mask\_getexp\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a);

VGETEXPPH \_\_m512h \_mm512\_maskz\_getexp\_ph (\_\_mmask32 k, \_\_m512h a);

VGETEXPPH \_\_m512h \_mm512\_getexp\_round\_ph (\_\_m512h a, const int sae);

VGETEXPPH \_\_m512h \_mm512\_mask\_getexp\_round\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, const int sae);

VGETEXPPH \_\_m512h \_mm512\_maskz\_getexp\_round\_ph (\_\_mmask32 k, \_\_m512h a, const int sae);

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

## VGETEXPPS—Convert Exponents of Packed Single Precision Floating-Point Values to Single Precision Floating-Point Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 42 /r<br>VGETEXPPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst      | A         | V/V                          | AVX512VL<br>AVX512F      | Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register. |
| EVEX.256.66.0F38.W0 42 /r<br>VGETEXPPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst      | A         | V/V                          | AVX512VL<br>AVX512F      | Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register. |
| EVEX.512.66.0F38.W0 42 /r<br>VGETEXPPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{sae} | A         | V/V                          | AVX512F                  | Convert the exponent of packed single-precision floating-point values in the source operand to single-precision floating-point results representing unbiased integer exponents and stores the results in the destination register. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

Extracts the biased exponents from the normalized single-precision floating-point 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 floating-point value and written to the corresponding dword elements of the destination operand (the first operand) as single-precision floating-point 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 floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-7.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **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 floating-point exceptions.

Table 5-7. VGETEXPPS/SS Special Cases

| Input Operand      | Result                         | Comments   |
|--------------------|--------------------------------|--|
| src1 = NaN         | QNaN(src1)                     | If (SRC = SNaN) then #IE<br>If (SRC = denormal) then #DE |
| $0 <  src1  < INF$ | $\text{floor}(\log_2( src1 ))$ |  |
| $ src1  = +INF$    | +INF                           |  |
| $ src1  = 0$       | -INF                           |  |

Figure 5-14 illustrates the VGETEXPPS functionality on input values with normalized representation.

|                                 | 31 | 30  | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22       | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |   |
|---------------------------------|----|-----|----|----|----|----|----|----|----|----------|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
|                                 | s  | exp |    |    |    |    |    |    |    | Fraction |    |    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |
| Src = 2 <sup>-1</sup>           | 0  | 1   | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0        | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SAR Src, 23 = 080h              | 0  | 0   | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0        | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| -Bias                           | 1  | 1   | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1        | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |   |
| Tmp - Bias = 1                  | 0  | 0   | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0        | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |   |
| Cvt_P12PS(01h) = 2 <sup>0</sup> | 0  | 0   | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0        | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |   |

Figure 5-14. VGETEXPPS Functionality On Normal Input values

### Operation

NormalizeExpTinySPFP(SRC[31:0])

```
{
    // Jbit is the hidden integral bit of a floating-point 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 floating-point number
}
}

```

**VGETEXPPS (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 32

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

THEN

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

THEN

DEST[i+31:i] :=

ConvertExpSPFP(SRC[31:0])

ELSE

DEST[i+31:i] :=

ConvertExpSPFP(SRC[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

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

VGETEXPPS \_\_m512 \_\_mm512\_getexp\_ps(\_\_m512 a);

VGETEXPPS \_\_m512 \_\_mm512\_mask\_getexp\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a);

VGETEXPPS \_\_m512 \_\_mm512\_maskz\_getexp\_ps(\_\_mmask16 k, \_\_m512 a);

VGETEXPPS \_\_m512 \_\_mm512\_getexp\_round\_ps(\_\_m512 a, int sae);

VGETEXPPS \_\_m512 \_\_mm512\_mask\_getexp\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, int sae);

VGETEXPPS \_\_m512 \_\_mm512\_maskz\_getexp\_round\_ps(\_\_mmask16 k, \_\_m512 a, int sae);

VGETEXPPS \_\_m256 \_\_mm256\_getexp\_ps(\_\_m256 a);

VGETEXPPS \_\_m256 \_\_mm256\_mask\_getexp\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VGETEXPPS \_\_m256 \_\_mm256\_maskz\_getexp\_ps(\_\_mmask8 k, \_\_m256 a);

VGETEXPPS \_\_m128 \_\_mm\_getexp\_ps(\_\_m128 a);

VGETEXPPS \_\_m128 \_\_mm\_mask\_getexp\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a);

VGETEXPPS \_\_m128 \_\_mm\_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 Double Precision Floating-Point Value to Double Precision Floating-Point Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.66.0F38.W1 43 /r<br>VGETEXPSD xmm1 {k1}{z},<br>xmm2, xmm3/m64{sae} | A         | V/V                          | AVX512F                  | Convert the biased exponent (bits 62:52) of the low double precision floating-point value in xmm3/m64 to a double precision floating-point 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Extracts the biased exponent from the normalized double precision floating-point 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 floating-point value and written to the destination operand (the first operand) as double precision floating-point 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 floating-point 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 **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, int sae);  
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.”

## VGETEXPSH—Convert Exponents of Scalar FP16 Values to FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|---|-----------|------------------------------|-----------------------|--|
| EVEEX.LLIG.66.MAP6.WO 43 /r<br>VGETEXPSH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {sae} | A         | V/V                          | AVX512-FP16           | Convert the exponent of FP16 values in the low word of the source operand to FP16 results representing unbiased integer exponents, and stores the results in the low word of the destination register subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction extracts the biased exponents from the normalized FP16 representation of the low word element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to an unbiased negative integer value. The integer value of the unbiased exponent is converted to an FP16 value and written to the low word element of the destination operand (the first operand) as an FP16 number. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

Each GETEXP operation converts the exponent value into a floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-6.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTSH). Thus, the VGETEXPSH instruction does not require software to handle SIMD floating-point exceptions.

### Operation

VGETEXPSH dest{k1}, src1, src2

IF k1[0] or \*no writemask\*:

DEST.fp16[0] := getexp\_fp16(src2.fp16[0]) // see VGETEXPPH

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]

DEST[MAXVL-1:128] := 0

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

VGETEXPSH \_\_m128h \_\_mm\_getexp\_round\_sh (\_\_m128h a, \_\_m128h b, const int sae);

VGETEXPSH \_\_m128h \_\_mm\_mask\_getexp\_round\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b, const int sae);

VGETEXPSH \_\_m128h \_\_mm\_maskz\_getexp\_round\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, const int sae);

VGETEXPSH \_\_m128h \_\_mm\_getexp\_sh (\_\_m128h a, \_\_m128h b);

VGETEXPSH \_\_m128h \_\_mm\_mask\_getexp\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VGETEXPSH \_\_m128h \_\_mm\_maskz\_getexp\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

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

## VGETEXPSS—Convert Exponents of Scalar Single Precision Floating-Point Value to Single Precision Floating-Point Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.66.0F38.W0 43 /r<br>VGETEXPSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae} | A         | V/V                          | AVX512F                  | Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a single-precision floating-point value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Extracts the biased exponent from the normalized single-precision floating-point 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 floating-point value and written to the destination operand (the first operand) as single-precision floating-point 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 floating-point number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-7.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **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 floating-point exceptions.

### Operation

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

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

#### VGETEXPSS (EVEX encoded version)

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

```
DEST[127:32] := SRC1[127:32]  
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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W1 26 /r ib<br>VGETMANTPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512F      | Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask. |
| EVEX.256.66.0F3A.W1 26 /r ib<br>VGETMANTPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512F      | Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask. |
| EVEX.512.66.0F3A.W1 26 /r ib<br>VGETMANTPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae},<br>imm8 | A         | V/V                          | AVX512F                  | Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | imm8      | N/A       |

### Description

Convert double precision floating values in the source operand (the second operand) to double precision floating-point 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.

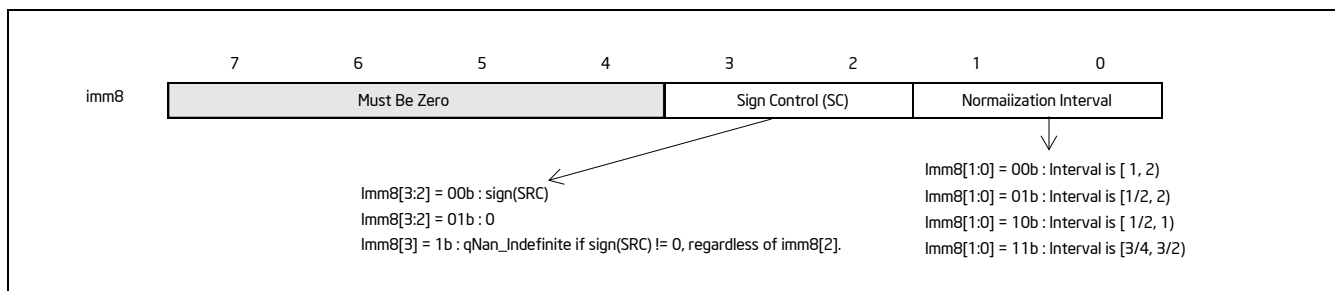


Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input double precision floating-point 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 double precision floating-point 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 5-8 when dealing with floating-point special numbers.

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

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

**Table 5-8. GetMant() Special Float Values Behavior**

| Input     | Result   | Exceptions / Comments                            |
|-----------|--|--|
| NaN       | QNaN(SRC)  | Ignore <i>interv</i><br>If (SRC = SNaN) then #IE |
| $+\infty$ | 1.0  | Ignore <i>interv</i>                             |
| +0        | 1.0  | Ignore <i>interv</i>                             |
| -0        | IF (SC[0]) THEN +1.0<br>ELSE -1.0  | Ignore <i>interv</i>                             |
| $-\infty$ | IF (SC[1]) THEN {QNaN_Indefinite}<br>ELSE {<br>IF (SC[0]) THEN +1.0<br>ELSE -1.0 | Ignore <i>interv</i><br>If (SC[1]) then #IE      |
| negative  | SC[1] ? QNaN_Indefinite : Getmant(SRC) <sup>1</sup>                              | If (SC[1]) then #IE                              |

#### NOTES:

1. In case SC[1]==0, the sign of Getmant(SRC) is declared according to SC[0].

#### Operation

```
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 sign_control[1]:
    MXCSR.IE := 1
    return QNaN_Indefinite

```

```

if denormal:
    jbit := 0
    dst.exp := bias
    while jbit = 0:
        jbit := dst.fraction[51]
        dst.fraction := dst.fraction << 1
        dst.exp := dst.exp - 1
    MXCSR.DE := 1

```

```

unbiased_exp := dst.exp - bias
odd_exp := unbiased_exp[0]
signaling_bit := dst.fraction[51]
if normalization_interval = 0b00:
    dst.exp := bias
else if normalization_interval = 0b01:
    dst.exp := odd_exp ? bias-1 : bias
else if normalization_interval = 0b10:
    dst.exp := bias-1
else if normalization_interval = 0b11:
    dst.exp := signaling_bit ? bias-1 : bias
return dst

```

**VGETMANTPD (EVEX Encoded Versions)**

VGETMANTPD dest{k1}, src, imm8

VL = 128, 256, or 512

KL := VL / 64

sign\_control := imm8[3:2]

normalization\_interval := imm8[1:0]

FOR i := 0 to KL-1:

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

IF SRC is memory and (EVEX.b = 1):

tsrc := src.double[0]

ELSE:

tsrc := src.double[i]

DEST.double[i] := getmant\_fp64(tsrc, sign\_control, normalization\_interval)

ELSE IF \*zeroing\*:

DEST.double[i] := 0

//else DEST.double[i] remains unchanged

DEST[MAX\_VL-1:VL] := 0



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

```

VGETMANTPD __m512d _mm512_getmant_pd( __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_maskz_getmant_pd( __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_getmant_round_pd( __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_maskz_getmant_round_pd( __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d _mm256_getmant_pd( __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_maskz_getmant_pd( __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_getmant_pd( __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_maskz_getmant_pd( __mmask8 k, __m128d a, enum intv, enum sgn);

```

**SIMD Floating-Point Exceptions**

Denormal, Invalid.

**Other Exceptions**

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

Additionally:

#UD                    If EVEX.vvvv != 1111B.

## VGETMANTPH—Extract FP16 Vector of Normalized Mantissas from FP16 Vector

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.0F3A.W0 26 /r /ib<br>VGETMANTPH xmm1{k1}{z}, xmm2/<br>m128/m16bcst, imm8       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Get normalized mantissa from FP16 vector<br>xmm2/m128/m16bcst and store the result in<br>xmm1, using imm8 for sign control and mantissa<br>interval normalization, subject to writemask k1. |
| EVEX.256.NP.0F3A.W0 26 /r /ib<br>VGETMANTPH ymm1{k1}{z}, ymm2/<br>m256/m16bcst, imm8       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Get normalized mantissa from FP16 vector<br>ymm2/m256/m16bcst and store the result in<br>ymm1, using imm8 for sign control and mantissa<br>interval normalization, subject to writemask k1. |
| EVEX.512.NP.0F3A.W0 26 /r /ib<br>VGETMANTPH zmm1{k1}{z}, zmm2/<br>m512/m16bcst {sae}, imm8 | A         | V/V                          | AVX512-FP16             | Get normalized mantissa from FP16 vector<br>zmm2/m512/m16bcst and store the result in<br>zmm1, using imm8 for sign control and mantissa<br>interval normalization, subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | imm8 (r)  | N/A       |

### Description

This instruction converts the FP16 values in the source operand (the second operand) to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 5-9. 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 elements are updated according to the writemask.

**Table 5-9. imm8 Controls for VGETMANTPH/VGETMANTSH**

| imm8 Bits | Definition   |
|-----------|--|
| imm8[7:4] | Must be zero.  |
| imm8[3:2] | Sign Control (SC)<br>0b00: Sign(SRC)<br>0b01: 0<br>0b1x: QNaN_Indefinite if sign(SRC)≠0  |
| imm8[1:0] | Interv<br>0b00: Interval is [1, 2)<br>0b01: Interval is [1/2, 2)<br>0b10: Interval is [1/2, 1)<br>0b11: Interval is [3/4, 3/2) |

For each input FP16 value x, The conversion operation is:

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

where:

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

Unbiased exponent  $k$  depends on the interval range defined by `interv` and whether the exponent of the source is even or odd. The sign of the final result is determined by the sign control and the source sign and the leading fraction bit.

The encoded value of `imm8[1:0]` and sign control are shown in Table 5-9.

Each converted FP16 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 5-10 when dealing with floating-point special numbers.

**Table 5-10. GetMant() Special Float Values Behavior**

| Input     | Result   | Exceptions / Comments                                |
|-----------|--|--|
| NaN       | QNaN(SRC)  | Ignore <i>interv</i> .<br>If (SRC = SNaN), then #IE. |
| $+\infty$ | 1.0  | Ignore <i>interv</i> .                               |
| +0        | 1.0  | Ignore <i>interv</i> .                               |
| -0        | IF (SC[0]) THEN +1.0<br>ELSE -1.0  | Ignore <i>interv</i> .                               |
| $-\infty$ | IF (SC[1]) THEN {QNaN_Indefinite}<br>ELSE {<br>IF (SC[0]) THEN +1.0<br>ELSE -1.0 | Ignore <i>interv</i> .<br>If (SC[1]), then #IE.      |
| negative  | SC[1] ? QNaN_Indefinite : Getmant(SRC) <sup>1</sup>                              | If (SC[1]), then #IE.                                |

#### NOTES:

1. In case `SC[1]==0`, the sign of `Getmant(SRC)` is declared according to `SC[0]`.

#### Operation

```
def getmant_fp16(src, sign_control, normalization_interval):
    bias := 15
    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)
    denormal := (dst.exp = 0) and (dst.fraction != 0)
    infinity := (dst.exp = 0x1F) and (dst.fraction = 0)
    nan := (dst.exp = 0x1F) and (dst.fraction != 0)
    src_signaling := src.fraction[9]
    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          // set exponent to bias value
    while jbit = 0:
        jbit := dst.fraction[9]
        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[9]
    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

```

**VGETMANTPH dest{k1}, src, imm8**

VL = 128, 256 or 512

KL := VL/16

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.fp16[0]

ELSE:

tsrc := src.fp16[i]

DEST.fp16[i] := getmant\_fp16(tsrc, sign\_control, normalization\_interval)

ELSE IF \*zeroing\*:

DEST.fp16[i] := 0

//else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

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

```

VGETMANTPH __m128h _mm_getmant_ph (__m128h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_mask_getmant_ph (__m128h src, __mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm,
    _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m128h _mm_maskz_getmant_ph (__mmask8 k, __m128h a, _MM_MANTISSA_NORM_ENUM norm,
    _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_getmant_ph (__m256h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_mask_getmant_ph (__m256h src, __mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm,
    _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m256h _mm256_maskz_getmant_ph (__mmask16 k, __m256h a, _MM_MANTISSA_NORM_ENUM norm,
    _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm, _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_mask_getmant_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm,
    _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_maskz_getmant_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm,
    _MM_MANTISSA_SIGN_ENUM sign);
VGETMANTPH __m512h _mm512_getmant_round_ph (__m512h a, _MM_MANTISSA_NORM_ENUM norm,
    _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_mask_getmant_round_ph (__m512h src, __mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM
    norm, _MM_MANTISSA_SIGN_ENUM sign, const int sae);
VGETMANTPH __m512h _mm512_maskz_getmant_round_ph (__mmask32 k, __m512h a, _MM_MANTISSA_NORM_ENUM norm,
    _MM_MANTISSA_SIGN_ENUM sign, const int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

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

## VGETMANTPS—Extract Float32 Vector of Normalized Mantissas From Float32 Vector

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W0 26 /r ib<br>VGETMANTPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512F      | Get normalized mantissa from float32 vector<br>xmm2/m128/m32bcst and store the result in xmm1, using<br>imm8 for sign control and mantissa interval normalization,<br>under writemask. |
| EVEX.256.66.0F3A.W0 26 /r ib<br>VGETMANTPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512F      | Get normalized mantissa from float32 vector<br>ymm2/m256/m32bcst and store the result in ymm1, using<br>imm8 for sign control and mantissa interval normalization,<br>under writemask. |
| EVEX.512.66.0F3A.W0 26 /r ib<br>VGETMANTPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{sae},<br>imm8 | A         | V/V                          | AVX512F                  | Get normalized mantissa from float32 vector<br>zmm2/m512/m32bcst and store the result in zmm1, using<br>imm8 for sign control and mantissa interval normalization,<br>under writemask. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | imm8      | N/A       |

### Description

Convert single-precision floating values in the source operand (the second operand) to single-precision floating-point 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 single-precision floating-point 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 single-precision floating-point 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 5-8 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 Mantissa From Float64 Scalar

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.66.0F3A.W1 27 /r ib<br>VGETMANTSD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{sae}, imm8 | A         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Convert the double precision floating values in the low quadword element of the second source operand (the third operand) to double precision floating-point value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

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

where:

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

Unbiased exponent k can be either 0 or -1, depending on the interval range defined by interv, the range of the significand and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted double precision floating-point 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 5-8 when dealing with floating-point special numbers.

If writemasking is used, the low quadword element of the destination operand is conditionally updated depending on the value of writemask register k1. If writemasking is not used, the low quadword element of the destination operand is unconditionally updated.

**Operation**

// getmant\_fp64(src, sign\_control, normalization\_interval) is defined in the operation section of VGETMANTPD

**VGETMANTSD (EVEX encoded version)**

```

SignCtrl[1:0] := IMM8[3:2];
Interv[1:0] := IMM8[1:0];
IF k1[0] OR *no writemask*
  THEN DEST[63:0] :=
    getmant_fp64(src, sign_control, normalization_interval)
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      DEST[63:0] := 0
    FI
  FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

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

```

VGETMANTSD __m128d __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, __m128 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."

## VGETMANTSH—Extract FP16 of Normalized Mantissa from FP16 Scalar

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|--|-----------|------------------------------|-----------------------|--|
| EVEEX.LLIG.NP.OF3A.WO 27 /r /ib<br>VGETMANTSH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {sae}, imm8 | A         | V/V                          | AVX512-FP16           | Extract the normalized mantissa of the low FP16 element in xmm3/m16 using imm8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 subject to writemask k1 and merge with the other elements of xmm2. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | imm8 (r)  |

### Description

This instruction converts the FP16 value in the low element of the second source operand to FP16 values with the mantissa normalization and sign control specified by the imm8 byte, see Table 5-9. The converted result is written to the low element of the destination 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.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For each input FP16 value  $x$ , The conversion operation is:

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

where:

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

Unbiased exponent  $k$  depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by the sign control and the source sign and the leading fraction bit.

The encoded value of imm8[1:0] and sign control are shown in Table 5-9.

Each converted FP16 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 5-10 when dealing with floating-point special numbers.

### Operation

**VGETMANTSH dest{k1}, src1, src2, imm8**

sign\_control := imm8[3:2]

normalization\_interval := imm8[1:0]

IF k1[0] or \*no writemask\*:

```
dest.fp16[0] := getmant_fp16(src2.fp16[0], // see VGETMANTPH
    sign_control,
    normalization_interval)
```

ELSE IF \*zeroing\*:

```
dest.fp16[0] := 0
```

//else dest.fp16[0] remains unchanged

```
DEST[127:16] := src1[127:16]
```

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

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

VGETMANTSH \_\_m128h\_mm\_getmant\_round\_sh (\_\_m128h a, \_\_m128h b, \_MM\_MANTISSA\_NORM\_ENUM norm, \_MM\_MANTISSA\_SIGN\_ENUM sign, const int sae);

VGETMANTSH \_\_m128h\_mm\_mask\_getmant\_round\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b, \_MM\_MANTISSA\_NORM\_ENUM norm, \_MM\_MANTISSA\_SIGN\_ENUM sign, const int sae);

VGETMANTSH \_\_m128h\_mm\_maskz\_getmant\_round\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, \_MM\_MANTISSA\_NORM\_ENUM norm, \_MM\_MANTISSA\_SIGN\_ENUM sign, const int sae);

VGETMANTSH \_\_m128h\_mm\_getmant\_sh (\_\_m128h a, \_\_m128h b, \_MM\_MANTISSA\_NORM\_ENUM norm, \_MM\_MANTISSA\_SIGN\_ENUM sign);

VGETMANTSH \_\_m128h\_mm\_mask\_getmant\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b, \_MM\_MANTISSA\_NORM\_ENUM norm, \_MM\_MANTISSA\_SIGN\_ENUM sign);

VGETMANTSH \_\_m128h\_mm\_maskz\_getmant\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, \_MM\_MANTISSA\_NORM\_ENUM norm, \_MM\_MANTISSA\_SIGN\_ENUM sign);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

## VGETMANTSS—Extract Float32 Vector of Normalized Mantissa From Float32 Scalar

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEEX.LLIG.66.0F3A.W0 27 /r ib<br>VGETMANTSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae}, imm8 | A         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2      | Operand 3     | Operand 4 |
|-------|---------------|---------------|----------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

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

The conversion operation is:

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

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 single-precision floating-point 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 5-8 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[127:32] := SRC1[127:32]
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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| VEX.256.66.0F3A.W0 18 /r ib<br>VINSERTF128 ymm1, ymm2,<br>xmm3/m128, imm8           | A          | V/V                          | AVX                      | Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.                                     |
| EVEX.256.66.0F3A.W0 18 /r ib<br>VINSERTF32X4 ymm1 {k1}{z}, ymm2,<br>xmm3/m128, imm8 | C          | V/V                          | AVX512VL<br>AVX512F      | Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1. |
| EVEX.512.66.0F3A.W0 18 /r ib<br>VINSERTF32X4 zmm1 {k1}{z}, zmm2,<br>xmm3/m128, imm8 | C          | V/V                          | AVX512F                  | Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.256.66.0F3A.W1 18 /r ib<br>VINSERTF64X2 ymm1 {k1}{z}, ymm2,<br>xmm3/m128, imm8 | B          | V/V                          | AVX512VL<br>AVX512DQ     | Insert 128 bits of packed double precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1. |
| EVEX.512.66.0F3A.W1 18 /r ib<br>VINSERTF64X2 zmm1 {k1}{z}, zmm2,<br>xmm3/m128, imm8 | B          | V/V                          | AVX512DQ                 | Insert 128 bits of packed double precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.512.66.0F3A.W0 1A /r ib<br>VINSERTF32X8 zmm1 {k1}{z}, zmm2,<br>ymm3/m256, imm8 | D          | V/V                          | AVX512DQ                 | Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.512.66.0F3A.W1 1A /r ib<br>VINSERTF64X4 zmm1 {k1}{z}, zmm2,<br>ymm3/m256, imm8 | C          | V/V                          | AVX512F                  | Insert 256 bits of packed double precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | imm8      |
| B     | Tuple2     | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |
| C     | Tuple4     | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |
| D     | Tuple8     | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### 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 a 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: TMP\_DEST[127:0] := SRC2[127:0]

1: TMP\_DEST[255:128] := SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] := SRC2[127:0]

01: TMP\_DEST[255:128] := SRC2[127:0]

10: TMP\_DEST[383:256] := SRC2[127:0]

11: TMP\_DEST[511:384] := SRC2[127:0]

ESAC.

FI;

FOR j := 0 TO KL-1

i := j \* 32

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

THEN DEST[i+31:i] := 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: TMP\_DEST[127:0] := SRC2[127:0]

1: TMP\_DEST[255:128] := SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] := SRC2[127:0]

01: TMP\_DEST[255:128] := SRC2[127:0]

10: TMP\_DEST[383:256] := SRC2[127:0]

11: TMP\_DEST[511:384] := SRC2[127:0]

ESAC.

FI;

FOR j := 0 TO KL-1

i := j \* 64

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

THEN DEST[i+63:i] := 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

```

**VINSERTF32x8 (EVEX.U1.512 encoded version)**

```

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

```

```

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

```

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

```

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

```

```

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

```

**VINSERTF128 (VEX encoded version)**

```

TEMP[255:0] := SRC1[255:0]
CASE (imm8[0]) OF
  0: TEMP[127:0] := SRC2[127:0]
  1: TEMP[255:128] := SRC2[127:0]
ESAC
DEST := TEMP

```

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

```

VINSERTF32x4 __m512 __mm512_insertf32x4(__m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 __mm512_mask_insertf32x4(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 __mm512_maskz_insertf32x4(__mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_insertf32x4(__m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_mask_insertf32x4(__m256 s, __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 __mm256_maskz_insertf32x4(__mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x8 __m512 __mm512_insertf32x8(__m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 __mm512_mask_insertf32x8(__m512 s, __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 __mm512_maskz_insertf32x8(__mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF64x2 __m512d __mm512_insertf64x2(__m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d __mm512_mask_insertf64x2(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d __mm512_maskz_insertf64x2(__mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m256d __mm256_insertf64x2(__m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d __mm256_mask_insertf64x2(__m256d s, __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d __mm256_maskz_insertf64x2(__mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x4 __m512d __mm512_insertf64x4(__m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d __mm512_mask_insertf64x4(__m512d s, __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d __mm512_maskz_insertf64x4(__mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF128 __m256 __mm256_insertf128_ps(__m256 a, __m128 b, int offset);
VINSERTF128 __m256d __mm256_insertf128_pd(__m256d a, __m128d b, int offset);
VINSERTF128 __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."

## VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>Bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| VEX.256.66.0F3A.W0 38 /r ib<br>VINSERTI128 ymm1, ymm2,<br>xmm3/m128, imm8           | A          | V/V                          | AVX2                     | Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.  |
| EVEX.256.66.0F3A.W0 38 /r ib<br>VINSERTI32X4 ymm1 {k1}{z}, ymm2,<br>xmm3/m128, imm8 | C          | V/V                          | AVX512VL<br>AVX512F      | Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1. |
| EVEX.512.66.0F3A.W0 38 /r ib<br>VINSERTI32X4 zmm1 {k1}{z}, zmm2,<br>xmm3/m128, imm8 | C          | V/V                          | AVX512F                  | Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.256.66.0F3A.W1 38 /r ib<br>VINSERTI64X2 ymm1 {k1}{z}, ymm2,<br>xmm3/m128, imm8 | B          | V/V                          | AVX512VL<br>AVX512DQ     | Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.   |
| EVEX.512.66.0F3A.W1 38 /r ib<br>VINSERTI64X2 zmm1 {k1}{z}, zmm2,<br>xmm3/m128, imm8 | B          | V/V                          | AVX512DQ                 | Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.   |
| EVEX.512.66.0F3A.W0 3A /r ib<br>VINSERTI32X8 zmm1 {k1}{z}, zmm2,<br>ymm3/m256, imm8 | D          | V/V                          | AVX512DQ                 | Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1. |
| EVEX.512.66.0F3A.W1 3A /r ib<br>VINSERTI64X4 zmm1 {k1}{z}, zmm2,<br>ymm3/m256, imm8 | C          | V/V                          | AVX512F                  | Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.   |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | imm8      |
| B     | Tuple2     | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |
| C     | Tuple4     | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |
| D     | Tuple8     | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### 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 a 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: TMP\_DEST[127:0] := SRC2[127:0]

1: TMP\_DEST[255:128] := SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] := SRC2[127:0]

01: TMP\_DEST[255:128] := SRC2[127:0]

10: TMP\_DEST[383:256] := SRC2[127:0]

11: TMP\_DEST[511:384] := SRC2[127:0]

ESAC.

FI;

FOR j := 0 TO KL-1

i := j \* 32

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

THEN DEST[i+31:i] := 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

#### VINSERTI64x2 (EVEX encoded versions)

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

TEMP\_DEST[VL-1:0] := SRC1[VL-1:0]

IF VL = 256

CASE (imm8[0]) OF

0: TMP\_DEST[127:0] := SRC2[127:0]

1: TMP\_DEST[255:128] := SRC2[127:0]

ESAC.

FI;

IF VL = 512

CASE (imm8[1:0]) OF

00: TMP\_DEST[127:0] := SRC2[127:0]

01: TMP\_DEST[255:128] := SRC2[127:0]

10: TMP\_DEST[383:256] := SRC2[127:0]

11: TMP\_DEST[511:384] := SRC2[127:0]

ESAC.

FI;

FOR j := 0 TO KL-1

i := j \* 64

```

IF k1[j] OR *no writemask*
  THEN DEST[i+63:i] := 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

```

**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]
  1: TEMP[255:128] := SRC2[127:0]
ESAC
DEST := TEMP
```

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

```
VINSERTI32x4 __mm512i_inserti32x4(__m512i a, __m128i b, int imm);
VINSERTI32x4 __mm512i_mask_inserti32x4(__m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 __mm512i_maskz_inserti32x4(__mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 __m256i_mm256_inserti32x4(__m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i_mm256_mask_inserti32x4(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i_mm256_maskz_inserti32x4(__mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x8 __m512i_mm512_inserti32x8(__m512i a, __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);
VINSERTI64x2 __m256i_mm256_inserti64x2(__m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i_mm256_mask_inserti64x2(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i_mm256_maskz_inserti64x2(__mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x4 __mm512i_inserti64x4(__m512i a, __m256i b, int imm);
VINSERTI64x4 __mm512i_mask_inserti64x4(__m512i s, __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI64x4 __mm512i_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

| Opcode/<br>Instruction                                   | Op/<br>En | 64/32-<br>bit Mode | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|--------------------|--------------------------|---|
| VEX.128.66.0F38.W0 2C /r<br>VMASKMOVPS xmm1, xmm2, m128  | RV<br>M   | V/V                | AVX                      | Conditionally load packed single-precision values from m128 using mask in xmm2 and store in xmm1. |
| VEX.256.66.0F38.W0 2C /r<br>VMASKMOVPS ymm1, ymm2, m256  | RV<br>M   | V/V                | AVX                      | Conditionally load packed single-precision values from m256 using mask in ymm2 and store in ymm1. |
| VEX.128.66.0F38.W0 2D /r<br>VMASKMOVDPD xmm1, xmm2, m128 | RV<br>M   | V/V                | AVX                      | Conditionally load packed double precision values from m128 using mask in xmm2 and store in xmm1. |
| VEX.256.66.0F38.W0 2D /r<br>VMASKMOVDPD ymm1, ymm2, m256 | RV<br>M   | V/V                | AVX                      | Conditionally load packed double precision values from m256 using mask in ymm2 and store in ymm1. |
| VEX.128.66.0F38.W0 2E /r<br>VMASKMOVPS m128, xmm1, xmm2  | MV<br>R   | V/V                | AVX                      | Conditionally store packed single-precision values from xmm2 using mask in xmm1.                  |
| VEX.256.66.0F38.W0 2E /r<br>VMASKMOVPS m256, ymm1, ymm2  | MV<br>R   | V/V                | AVX                      | Conditionally store packed single-precision values from ymm2 using mask in ymm1.                  |
| VEX.128.66.0F38.W0 2F /r<br>VMASKMOVDPD m128, xmm1, xmm2 | MV<br>R   | V/V                | AVX                      | Conditionally store packed double precision values from xmm2 using mask in xmm1.                  |
| VEX.256.66.0F38.W0 2F /r<br>VMASKMOVDPD m256, ymm1, ymm2 | MV<br>R   | V/V                | AVX                      | Conditionally store packed double precision values from ymm2 using mask in ymm1.                  |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|---------------|--------------|---------------|-----------|
| RVM   | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| MVR   | ModRM:r/m (w) | VEX.vvvv (r) | ModRM:reg (r) | N/A       |

### 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 MASKMOVDPD), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm\_field, and the destination register is encoded in reg\_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg\_field, and the destination memory location is encoded in rm\_field.

## Operation

### VMASKMOVPS - 128-bit load

```
DEST[31:0] := IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] := IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] := IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] := IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[MAXVL-1:128] := 0
```

### VMASKMOVPS - 256-bit load

```
DEST[31:0] := IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] := IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] := IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] := IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] := IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] := IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] := IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] := IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

### VMASKMOVPD - 128-bit load

```
DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[MAXVL-1:128] := 0
```

### VMASKMOVPD - 256-bit load

```
DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] := IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] := IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

### VMASKMOVPS - 128-bit store

```
IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[63]) DEST[63:32] := SRC2[63:32]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]
IF (SRC1[127]) DEST[127:96] := SRC2[127:96]
```

### VMASKMOVPS - 256-bit store

```
IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[63]) DEST[63:32] := SRC2[63:32]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]
IF (SRC1[127]) DEST[127:96] := SRC2[127:96]
IF (SRC1[159]) DEST[159:128] := SRC2[159:128]
IF (SRC1[191]) DEST[191:160] := SRC2[191:160]
IF (SRC1[223]) DEST[223:192] := SRC2[223:192]
IF (SRC1[255]) DEST[255:224] := SRC2[255:224]
```



**VMASKMOVPD - 128-bit store**

```
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]
```

**VMASKMOVPD - 256-bit store**

```
IF (SRC1[63]) DEST[63:0] := SRC2[63:0]
IF (SRC1[127]) DEST[127:64] := SRC2[127:64]
IF (SRC1[191]) DEST[191:128] := SRC2[191:128]
IF (SRC1[255]) DEST[255:192] := SRC2[255:192]
```

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

```
__m256 _mm256_maskload_ps(float const *a, __m256i mask)
void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)
__m256d _mm256_maskload_pd(double *a, __m256i mask);
void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);
__m128 _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.

## VMAXPH—Return Maximum of Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.NP.MAP5.WO 5F /r<br>VMAXPH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Return the maximum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1. |
| EVEX.256.NP.MAP5.WO 5F /r<br>VMAXPH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Return the maximum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1. |
| EVEX.512.NP.MAP5.WO 5F /r<br>VMAXPH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Return the maximum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|---------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second 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 broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

### Operation

```
def MAX(SRC1, SRC2):
  IF (SRC1 = 0.0) and (SRC2 = 0.0):
    DEST := SRC2
  ELSE IF (SRC1 = NaN):
    DEST := SRC2
  ELSE IF (SRC2 = NaN):
    DEST := SRC2
  ELSE IF (SRC1 > SRC2):
    DEST := SRC1
  ELSE:
    DEST := SRC2
```

**VMAXPH dest, src1, src2**

VL = 128, 256 or 512

KL := VL/16

```

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF EVEX.b = 1:
      tsrc2 := SRC2.fp16[0]
    ELSE:
      tsrc2 := SRC2.fp16[j]
    DEST.fp16[j] := MAX(SRC1.fp16[j], tsrc2)
  ELSE IF *zeroing*:
    DEST.fp16[j] := 0
  // else dest.fp16[j] remains unchanged

```

DEST[MAXVL-1:VL] := 0

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

```

VMAXPH __m128h __mm_mask_max_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMAXPH __m128h __mm_maskz_max_ph (__mmask8 k, __m128h a, __m128h b);
VMAXPH __m128h __mm_max_ph (__m128h a, __m128h b);
VMAXPH __m256h __mm256_mask_max_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMAXPH __m256h __mm256_maskz_max_ph (__mmask16 k, __m256h a, __m256h b);
VMAXPH __m256h __mm256_max_ph (__m256h a, __m256h b);
VMAXPH __m512h __mm512_mask_max_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h __mm512_maskz_max_ph (__mmask32 k, __m512h a, __m512h b);
VMAXPH __m512h __mm512_max_ph (__m512h a, __m512h b);
VMAXPH __m512h __mm512_mask_max_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h __mm512_maskz_max_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMAXPH __m512h __mm512_max_round_ph (__m512h a, __m512h b, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

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

## VMAXSH—Return Maximum of Scalar FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEEX.LLIG.F3.MAP5.W0 5F /r<br>VMAXSH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {sae} | A         | V/V                          | AVX512-FP16           | Return the maximum low FP16 value between xmm3/m16 and xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the maximum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMAXSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

```
def MAX(SRC1, SRC2):
  IF (SRC1 = 0.0) and (SRC2 = 0.0):
    DEST := SRC2
  ELSE IF (SRC1 = NaN):
    DEST := SRC2
  ELSE IF (SRC2 = NaN):
    DEST := SRC2
  ELSE IF (SRC1 > SRC2):
    DEST := SRC1
  ELSE:
    DEST := SRC2
```

### VMAXSH dest, src1, src2

```
IF k1[0] OR *no writemask*:
  DEST.fp16[0] := MAX(SRC1.fp16[0], SRC2.fp16[0])
ELSE IF *zeroing*:
  DEST.fp16[0] := 0
// else dest.fp16[j] remains unchanged
```

```
DEST[127:16] := SRC1[127:16]
DEST[MAXVL-1:128] := 0
```

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

VMAXSH \_\_m128h \_mm\_mask\_max\_round\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b, int sae);  
VMAXSH \_\_m128h \_mm\_maskz\_max\_round\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, int sae);  
VMAXSH \_\_m128h \_mm\_max\_round\_sh (\_\_m128h a, \_\_m128h b, int sae);  
VMAXSH \_\_m128h \_mm\_mask\_max\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
VMAXSH \_\_m128h \_mm\_maskz\_max\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
VMAXSH \_\_m128h \_mm\_max\_sh (\_\_m128h a, \_\_m128h b);

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

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

## VMINPH—Return Minimum of Packed FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|-----------|------------------------------|-------------------------|--|
| EVEX.128.NP.MAP5.W0 5D /r<br>VMINPH xmm1{k1}{z}, xmm2, xmm3/<br>m128/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Return the minimum packed FP16 values between xmm2 and xmm3/m128/m16bcst and store the result in xmm1 subject to writemask k1. |
| EVEX.256.NP.MAP5.W0 5D /r<br>VMINPH ymm1{k1}{z}, ymm2, ymm3/<br>m256/m16bcst       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Return the minimum packed FP16 values between ymm2 and ymm3/m256/m16bcst and store the result in ymm1 subject to writemask k1. |
| EVEX.512.NP.MAP5.W0 5D /r<br>VMINPH zmm1{k1}{z}, zmm2, zmm3/<br>m512/m16bcst {sae} | A         | V/V                          | AVX512-FP16             | Return the minimum packed FP16 values between zmm2 and zmm3/m512/m16bcst and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|---------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a SIMD compare of the packed FP16 values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINPH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second 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 broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

### Operation

```
def MIN(SRC1, SRC2):
  IF (SRC1 = 0.0) and (SRC2 = 0.0):
    DEST := SRC2
  ELSE IF (SRC1 = NaN):
    DEST := SRC2
  ELSE IF (SRC2 = NaN):
    DEST := SRC2
  ELSE IF (SRC1 < SRC2):
    DEST := SRC1
  ELSE:
    DEST := SRC2
```

**VMINPH dest, src1, src2**

VL = 128, 256 or 512

KL := VL/16

```

FOR j := 0 TO KL-1:
  IF k1[j] OR *no writemask*:
    IF EVEX.b = 1:
      tsrc2 := SRC2.fp16[0]
    ELSE:
      tsrc2 := SRC2.fp16[j]
    DEST.fp16[j] := MIN(SRC1.fp16[j], tsrc2)
  ELSE IF *zeroing*:
    DEST.fp16[j] := 0
  // else dest.fp16[j] remains unchanged

```

DEST[MAXVL-1:VL] := 0

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

```

VMINPH __m128h __mm_mask_min_ph (__m128h src, __mmask8 k, __m128h a, __m128h b);
VMINPH __m128h __mm_maskz_min_ph (__mmask8 k, __m128h a, __m128h b);
VMINPH __m128h __mm_min_ph (__m128h a, __m128h b);
VMINPH __m256h __mm256_mask_min_ph (__m256h src, __mmask16 k, __m256h a, __m256h b);
VMINPH __m256h __mm256_maskz_min_ph (__mmask16 k, __m256h a, __m256h b);
VMINPH __m256h __mm256_min_ph (__m256h a, __m256h b);
VMINPH __m512h __mm512_mask_min_ph (__m512h src, __mmask32 k, __m512h a, __m512h b);
VMINPH __m512h __mm512_maskz_min_ph (__mmask32 k, __m512h a, __m512h b);
VMINPH __m512h __mm512_min_ph (__m512h a, __m512h b);
VMINPH __m512h __mm512_mask_min_round_ph (__m512h src, __mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h __mm512_maskz_min_round_ph (__mmask32 k, __m512h a, __m512h b, int sae);
VMINPH __m512h __mm512_min_round_ph (__m512h a, __m512h b, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

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

## VMINSH—Return Minimum Scalar FP16 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F3.MAP5.W0 5D /r<br>VMINSH xmm1{k1}{z}, xmm2, xmm3/<br>m16 {sae} | A         | V/V                          | AVX512-FP16           | Return the minimum low FP16 value between xmm3/m16 and xmm2. Stores the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a compare of the low packed FP16 values in the first source operand and the second source operand and returns the minimum value for the pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of VMINSH can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN, and OR.

EVEX encoded versions: The first source operand (the second 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 broadcast from a 16-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

```
def MIN(SRC1, SRC2):
  IF (SRC1 = 0.0) and (SRC2 = 0.0):
    DEST := SRC2
  ELSE IF (SRC1 = NaN):
    DEST := SRC2
  ELSE IF (SRC2 = NaN):
    DEST := SRC2
  ELSE IF (SRC1 < SRC2):
    DEST := SRC1
  ELSE:
    DEST := SRC2
```



**VMINSH dest, src1, src2**

```

IF k1[0] OR *no writemask*:
    DEST.fp16[0] := MIN(SRC1.fp16[0], SRC2.fp16[0])
ELSE IF *zeroing*:
    DEST.fp16[0] := 0
// else dest.fp16[j] remains unchanged

```

```
DEST[127:16] := SRC1[127:16]
```

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

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

```
VMINSH __m128h __mm_mask_min_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int sae);
```

```
VMINSH __m128h __mm_maskz_min_round_sh (__mmask8 k, __m128h a, __m128h b, int sae);
```

```
VMINSH __m128h __mm_min_round_sh (__m128h a, __m128h b, int sae);
```

```
VMINSH __m128h __mm_mask_min_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
```

```
VMINSH __m128h __mm_maskz_min_sh (__mmask8 k, __m128h a, __m128h b);
```

```
VMINSH __m128h __mm_min_sh (__m128h a, __m128h b);
```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

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

## VMOVSH—Move Scalar FP16 Value

| Opcode/<br>Instruction                                       | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F3.MAP5.WO 10 /r<br>VMOVSH xmm1{k1}{z}, m16        | A         | V/V                          | AVX512-FP16           | Move FP16 value from m16 to xmm1 subject to writemask k1.   |
| EVEX.LLIG.F3.MAP5.WO 11 /r<br>VMOVSH m16{k1}, xmm1           | B         | V/V                          | AVX512-FP16           | Move low FP16 value from xmm1 to m16 subject to writemask k1.   |
| EVEX.LLIG.F3.MAP5.WO 10 /r<br>VMOVSH xmm1{k1}{z}, xmm2, xmm3 | C         | V/V                          | AVX512-FP16           | Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |
| EVEX.LLIG.F3.MAP5.WO 11 /r<br>VMOVSH xmm1{k1}{z}, xmm2, xmm3 | D         | V/V                          | AVX512-FP16           | Move low FP16 values from xmm3 to xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|--------|---------------|---------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A           | N/A       |
| B     | Scalar | ModRM:r/m (w) | ModRM:reg (r) | N/A           | N/A       |
| C     | N/A    | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| D     | N/A    | ModRM:r/m (w) | VEX.vvvv (r)  | ModRM:reg (r) | N/A       |

### Description

This instruction moves a FP16 value to a register or memory location.

The two register-only forms are aliases and differ only in where their operands are encoded; this is a side effect of the encodings selected.

### Operation

#### VMOVSH dest, src (two operand load)

IF k1[0] or no writemask:

DEST.fp16[0] := SRC.fp16[0]

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

// ELSE DEST.fp16[0] remains unchanged

DEST[MAXVL:16] := 0

#### VMOVSH dest, src (two operand store)

IF k1[0] or no writemask:

DEST.fp16[0] := SRC.fp16[0]

// ELSE DEST.fp16[0] remains unchanged

**VMOVSH dest, src1, src2 (three operand copy)**

IF k1[0] or no writemask:

DEST.fp16[0] := SRC2.fp16[0]

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

// ELSE DEST.fp16[0] remains unchanged

DEST[127:16] := SRC1[127:16]

DEST[MAXVL:128] := 0

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

VMOVSH \_\_m128h \_\_mm\_load\_sh (void const\* mem\_addr);

VMOVSH \_\_m128h \_\_mm\_mask\_load\_sh (\_\_m128h src, \_\_mmask8 k, void const\* mem\_addr);

VMOVSH \_\_m128h \_\_mm\_maskz\_load\_sh (\_\_mmask8 k, void const\* mem\_addr);

VMOVSH \_\_m128h \_\_mm\_mask\_move\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VMOVSH \_\_m128h \_\_mm\_maskz\_move\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VMOVSH \_\_m128h \_\_mm\_move\_sh (\_\_m128h a, \_\_m128h b);

VMOVSH void \_\_mm\_mask\_store\_sh (void \* mem\_addr, \_\_mmask8 k, \_\_m128h a);

VMOVSH void \_\_mm\_store\_sh (void \* mem\_addr, \_\_m128h a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Table 2-51, “Type E5 Class Exception Conditions.”

**VMOVW—Move Word**

| Opcode/<br>Instruction                            | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description                     |
|---|-----------|------------------------------|-----------------------|---------------------------------|
| EVEX.128.66.MAP5.WIG 6E /r<br>VMOVW xmm1, reg/m16 | A         | V/V                          | AVX512-FP16           | Copy word from reg/m16 to xmm1. |
| EVEX.128.66.MAP5.WIG 7E /r<br>VMOVW reg/m16, xmm1 | B         | V/V                          | AVX512-FP16           | Copy word from xmm1 to reg/m16. |

**Instruction Operand Encoding**

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

**Description**

This instruction either (a) copies one word element from an XMM register to a general-purpose register or memory location or (b) copies one word element from a general-purpose register or memory location to an XMM register. When writing a general-purpose register, the lower 16-bits of the register will contain the word value. The upper bits of the general-purpose register are written with zeros.

**Operation****VMOVW dest, src (two operand load)**

DEST.word[0] := SRC.word[0]  
DEST[MAXVL:16] := 0

**VMOVW dest, src (two operand store)**

DEST.word[0] := SRC.word[0]  
// upper bits of GPR DEST are zeroed

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

VMOVW short \_\_mm\_cvtsi128\_si16 (\_\_m128i a);  
VMOVW \_\_m128i \_\_mm\_cvtsi16\_si128 (short a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instructions, see Table 2-57, "Type E9NF Class Exception Conditions."

## VMULPH—Multiply Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.MAP5.W0 59 /r<br>VMULPH xmm1{k1}{z}, xmm2, xmm3/<br>m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from xmm3/m128/<br>m16bcst to xmm2 and store the result in xmm1<br>subject to writemask k1. |
| EVEX.256.NP.MAP5.W0 59 /r<br>VMULPH ymm1{k1}{z}, ymm2, ymm3/<br>m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Multiply packed FP16 values from ymm3/m256/<br>m16bcst to ymm2 and store the result in ymm1<br>subject to writemask k1. |
| EVEX.512.NP.MAP5.W0 59 /r<br>VMULPH zmm1{k1}{z}, zmm2, zmm3/<br>m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Multiply packed FP16 values in zmm3/m512/<br>m16bcst with zmm2 and store the result in<br>zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|---------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction multiplies packed FP16 values from source operands and stores the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

### Operation

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

VL = 128, 256 or 512

KL := VL/16

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

    SET\_RM(EVEX.RC)

ELSE

    SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

    IF k1[j] OR \*no writemask\*:

        DEST.fp16[j] := SRC1.fp16[j] \* SRC2.fp16[j]

    ELSE IF \*zeroing\*:

        DEST.fp16[j] := 0

    // else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VMULPH (EVEX encoded versions) when src2 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

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

IF EVEX.b = 1:

DEST.fp16[j] := SRC1.fp16[j] \* SRC2.fp16[0]

ELSE:

DEST.fp16[j] := SRC1.fp16[j] \* SRC2.fp16[j]

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

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

VMULPH \_\_m128h \_mm\_mask\_mul\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VMULPH \_\_m128h \_mm\_maskz\_mul\_ph (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VMULPH \_\_m128h \_mm\_mul\_ph (\_\_m128h a, \_\_m128h b);

VMULPH \_\_m256h \_mm256\_mask\_mul\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a, \_\_m256h b);

VMULPH \_\_m256h \_mm256\_maskz\_mul\_ph (\_\_mmask16 k, \_\_m256h a, \_\_m256h b);

VMULPH \_\_m256h \_mm256\_mul\_ph (\_\_m256h a, \_\_m256h b);

VMULPH \_\_m512h \_mm512\_mask\_mul\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, \_\_m512h b);

VMULPH \_\_m512h \_mm512\_maskz\_mul\_ph (\_\_mmask32 k, \_\_m512h a, \_\_m512h b);

VMULPH \_\_m512h \_mm512\_mul\_ph (\_\_m512h a, \_\_m512h b);

VMULPH \_\_m512h \_mm512\_mask\_mul\_round\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, \_\_m512h b, int rounding);

VMULPH \_\_m512h \_mm512\_maskz\_mul\_round\_ph (\_\_mmask32 k, \_\_m512h a, \_\_m512h b, int rounding);

VMULPH \_\_m512h \_mm512\_mul\_round\_ph (\_\_m512h a, \_\_m512h b, int rounding);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal

**Other Exceptions**

EVEX-encoded instructions, see Table 2-46, "Type E2 Class Exception Conditions."

## VMULSH—Multiply Scalar FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|--|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.F3.MAP5.W0 59 /r<br>VMULSH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Multiply the low FP16 value in xmm3/m16 by low FP16 value in xmm2, and store the result in xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction multiplies the low FP16 value from the source operands and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

#### VMULSH (EVEX encoded versions)

IF EVEX.b = 1 and SRC2 is a register:

```
SET_RM(EVEX.RC)
```

ELSE

```
SET_RM(MXCSR.RC)
```

IF k1[0] OR \*no writemask\*:

```
DEST.fp16[0] := SRC1.fp16[0] * SRC2.fp16[0]
```

ELSE IF \*zeroing\*:

```
DEST.fp16[0] := 0
```

// else dest.fp16[0] remains unchanged

```
DEST[127:16] := SRC1[127:16]
```

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

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

```
VMULSH __m128h __mm_mask_mul_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
```

```
VMULSH __m128h __mm_maskz_mul_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
```

```
VMULSH __m128h __mm_mul_round_sh (__m128h a, __m128h b, int rounding);
```

```
VMULSH __m128h __mm_mask_mul_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
```

```
VMULSH __m128h __mm_maskz_mul_sh (__mmask8 k, __m128h a, __m128h b);
```

```
VMULSH __m128h __mm_mul_sh (__m128h a, __m128h b);
```

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

## VP2INTERSECTD/VP2INTERSECTQ—Compute Intersection Between DWORDS/QUADWORDS to a Pair of Mask Registers

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature Flag              | Description   |
|---|-----------|------------------------------|---------------------------------|---|
| EVEX.NDS.128.F2.0F38.W0 68 /r<br>VP2INTERSECTD k1+1, xmm2,<br>xmm3/m128/m32bcst | A         | V/V                          | AVX512VL<br>AVX512_VP2INTERSECT | 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.    |
| EVEX.NDS.256.F2.0F38.W0 68 /r<br>VP2INTERSECTD k1+1, ymm2,<br>ymm3/m256/m32bcst | A         | V/V                          | AVX512VL<br>AVX512_VP2INTERSECT | 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.    |
| EVEX.NDS.512.F2.0F38.W0 68 /r<br>VP2INTERSECTD k1+1, zmm2,<br>zmm3/m512/m32bcst | A         | V/V                          | AVX512F<br>AVX512_VP2INTERSECT  | 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.    |
| EVEX.NDS.128.F2.0F38.W1 68 /r<br>VP2INTERSECTQ k1+1, xmm2,<br>xmm3/m128/m64bcst | A         | V/V                          | AVX512VL<br>AVX512_VP2INTERSECT | 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. |
| EVEX.NDS.256.F2.0F38.W1 68 /r<br>VP2INTERSECTQ k1+1, ymm2,<br>ymm3/m256/m64bcst | A         | V/V                          | AVX512VL<br>AVX512_VP2INTERSECT | 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. |
| EVEX.NDS.512.F2.0F38.W1 68 /r<br>VP2INTERSECTQ k1+1, zmm2,<br>zmm3/m512/m64bcst | A         | V/V                          | AVX512F<br>AVX512_VP2INTERSECT  | 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. |

### Instruction Operand Encoding

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

### 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."

## VPBLEND—Blend Packed Dwords

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>-bit<br>Mode | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|-----------------------|--------------------------|---|
| VEX.128.66.0F3A.W0 02 /r ib<br>VPBLEND xmm1, xmm2, xmm3/m128, imm8 | RVM1      | V/V                   | AVX2                     | Select dwords from xmm2 and xmm3/m128 from mask specified in imm8 and store the values into xmm1. |
| VEX.256.66.0F3A.W0 02 /r ib<br>VPBLEND ymm1, ymm2, ymm3/m256, imm8 | RVM1      | V/V                   | AVX2                     | Select dwords from ymm2 and ymm3/m256 from mask specified in imm8 and store the values into ymm1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|---------------|--------------|---------------|-----------|
| RVM1  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### 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 dword in the destination is copied from the source. If a bit in the mask, corresponding to a dword, is "1", then the dword is copied, else the dword 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

#### VPBLEND (VEX.256 encoded version)

```

IF (imm8[0] == 1) THEN DEST[31:0] := SRC2[31:0]
ELSE DEST[31:0] := SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] := SRC2[63:32]
ELSE DEST[63:32] := SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] := SRC2[95:64]
ELSE DEST[95:64] := SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] := SRC2[127:96]
ELSE DEST[127:96] := SRC1[127:96]
IF (imm8[4] == 1) THEN DEST[159:128] := SRC2[159:128]
ELSE DEST[159:128] := SRC1[159:128]
IF (imm8[5] == 1) THEN DEST[191:160] := SRC2[191:160]
ELSE DEST[191:160] := SRC1[191:160]
IF (imm8[6] == 1) THEN DEST[223:192] := SRC2[223:192]
ELSE DEST[223:192] := SRC1[223:192]
IF (imm8[7] == 1) THEN DEST[255:224] := SRC2[255:224]
ELSE DEST[255:224] := SRC1[255:224]

```

**VPBLEND (VEX.128 encoded version)**

```

IF (imm8[0] == 1) THEN DEST[31:0] := SRC2[31:0]
ELSE DEST[31:0] := SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] := SRC2[63:32]
ELSE DEST[63:32] := SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] := SRC2[95:64]
ELSE DEST[95:64] := SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] := SRC2[127:96]
ELSE DEST[127:96] := SRC1[127:96]
DEST[MAXVL-1:128] := 0

```

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

VPBLEND: `__m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)`

VPBLEND: `__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/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 66 /r<br>VPBLENDMB xmm1 {k1}{z},<br>xmm2, xmm3/m128 | A         | V/V                          | AVX512VL<br>AVX512BW     | Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask. |
| EVEX.256.66.0F38.W0 66 /r<br>VPBLENDMB ymm1 {k1}{z},<br>ymm2, ymm3/m256 | A         | V/V                          | AVX512VL<br>AVX512BW     | Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask. |
| EVEX.512.66.0F38.W0 66 /r<br>VPBLENDMB zmm1 {k1}{z},<br>zmm2, zmm3/m512 | A         | V/V                          | AVX512BW                 | Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask. |
| EVEX.128.66.0F38.W1 66 /r<br>VPBLENDMW xmm1 {k1}{z},<br>xmm2, xmm3/m128 | A         | V/V                          | AVX512VL<br>AVX512BW     | Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask. |
| EVEX.256.66.0F38.W1 66 /r<br>VPBLENDMW ymm1 {k1}{z},<br>ymm2, ymm3/m256 | A         | V/V                          | AVX512VL<br>AVX512BW     | Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask. |
| EVEX.512.66.0F38.W1 66 /r<br>VPBLENDMW zmm1 {k1}{z},<br>zmm2, zmm3/m512 | A         | V/V                          | AVX512BW                 | Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask. |

### Instruction Operand Encoding

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

### 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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 64 /r<br>VPBLENDMD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst | A          | V/V                          | AVX512VL<br>AVX512F      | Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask. |
| EVEX.256.66.0F38.W0 64 /r<br>VPBLENDMD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | A          | V/V                          | AVX512VL<br>AVX512F      | Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask. |
| EVEX.512.66.0F38.W0 64 /r<br>VPBLENDMD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | A          | V/V                          | AVX512F                  | Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask. |
| EVEX.128.66.0F38.W1 64 /r<br>VPBLENDMQ xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | A          | V/V                          | AVX512VL<br>AVX512F      | Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.     |
| EVEX.256.66.0F38.W1 64 /r<br>VPBLENDMQ ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | A          | V/V                          | AVX512VL<br>AVX512F      | Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.     |
| EVEX.512.66.0F38.W1 64 /r<br>VPBLENDMQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | A          | V/V                          | AVX512F                  | Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.     |

### Instruction Operand Encoding

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

### 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/<br>Instruction                                      | Op/<br>En | 64/32<br>bit<br>Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|---------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 7A /r<br>VPBROADCASTB xmm1 {k1}{z}, reg | A         | V/V                             | AVX512VL<br>AVX512BW     | Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.       |
| EVEX.256.66.0F38.W0 7A /r<br>VPBROADCASTB ymm1 {k1}{z}, reg | A         | V/V                             | AVX512VL<br>AVX512BW     | Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.       |
| EVEX.512.66.0F38.W0 7A /r<br>VPBROADCASTB zmm1 {k1}{z}, reg | A         | V/V                             | AVX512BW                 | Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.       |
| EVEX.128.66.0F38.W0 7B /r<br>VPBROADCASTW xmm1 {k1}{z}, reg | A         | V/V                             | AVX512VL<br>AVX512BW     | Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.       |
| EVEX.256.66.0F38.W0 7B /r<br>VPBROADCASTW ymm1 {k1}{z}, reg | A         | V/V                             | AVX512VL<br>AVX512BW     | Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.       |
| EVEX.512.66.0F38.W0 7B /r<br>VPBROADCASTW zmm1 {k1}{z}, reg | A         | V/V                             | AVX512BW                 | Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.       |
| EVEX.128.66.0F38.W0 7C /r<br>VPBROADCASTD xmm1 {k1}{z}, r32 | A         | V/V                             | AVX512VL<br>AVX512F      | Broadcast a 32-bit value from a GPR to all doublewords in the 128-bit destination subject to writemask k1. |
| EVEX.256.66.0F38.W0 7C /r<br>VPBROADCASTD ymm1 {k1}{z}, r32 | A         | V/V                             | AVX512VL<br>AVX512F      | Broadcast a 32-bit value from a GPR to all doublewords in the 256-bit destination subject to writemask k1. |
| EVEX.512.66.0F38.W0 7C /r<br>VPBROADCASTD zmm1 {k1}{z}, r32 | A         | V/V                             | AVX512F                  | Broadcast a 32-bit value from a GPR to all doublewords in the 512-bit destination subject to writemask k1. |
| EVEX.128.66.0F38.W1 7C /r<br>VPBROADCASTQ xmm1 {k1}{z}, r64 | A         | V/N.E. <sup>1</sup>             | AVX512VL<br>AVX512F      | Broadcast a 64-bit value from a GPR to all quadwords in the 128-bit destination subject to writemask k1.   |
| EVEX.256.66.0F38.W1 7C /r<br>VPBROADCASTQ ymm1 {k1}{z}, r64 | A         | V/N.E. <sup>1</sup>             | AVX512VL<br>AVX512F      | Broadcast a 64-bit value from a GPR to all quadwords in the 256-bit destination subject to writemask k1.   |
| EVEX.512.66.0F38.W1 7C /r<br>VPBROADCASTQ zmm1 {k1}{z}, r64 | A         | V/N.E. <sup>1</sup>             | AVX512F                  | Broadcast a 64-bit value from a GPR to all quadwords in the 512-bit destination subject to writemask k1.   |

### NOTES:

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

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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 __mm128_mask_set1_epi8(__m128i s, __mmask16 k, int a);
VPBROADCASTB __m128i __mm128_maskz_set1_epi8(__mmask16 k, int a);
VPBROADCASTD __m512i __mm512_mask_set1_epi32(__m512i s, __mmask16 k, int a);
VPBROADCASTD __m512i __mm512_maskz_set1_epi32(__mmask16 k, int a);
VPBROADCASTD __m256i __mm256_mask_set1_epi32(__m256i s, __mmask8 k, int a);
VPBROADCASTD __m256i __mm256_maskz_set1_epi32(__mmask8 k, int a);
VPBROADCASTD __m128i __mm128_mask_set1_epi32(__m128i s, __mmask8 k, int a);
VPBROADCASTD __m128i __mm128_maskz_set1_epi32(__mmask8 k, int 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, __mmask8 k, __int64 a);
VPBROADCASTQ __m256i __mm256_maskz_set1_epi64(__mmask8 k, __int64 a);
VPBROADCASTQ __m128i __mm128_mask_set1_epi64(__m128i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m128i __mm128_maskz_set1_epi64(__mmask8 k, __int64 a);
VPBROADCASTW __m512i __mm512_mask_set1_epi16(__m512i s, __mmask32 k, int a);
VPBROADCASTW __m512i __mm512_maskz_set1_epi16(__mmask32 k, int a);
VPBROADCASTW __m256i __mm256_mask_set1_epi16(__m256i s, __mmask16 k, int a);
VPBROADCASTW __m256i __mm256_maskz_set1_epi16(__mmask16 k, int a);
VPBROADCASTW __m128i __mm128_mask_set1_epi16(__m128i s, __mmask8 k, int a);
VPBROADCASTW __m128i __mm128_maskz_set1_epi16(__mmask8 k, int a);

```

**Exceptions**

EVEX-encoded instructions, see Table 2-55, "Type E7NM Class Exception Conditions."

Additionally:

#UD If EVEX.vvvv != 1111B.

## VPBROADCAST—Load Integer and Broadcast

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 78 /r<br>VPBROADCASTB xmm1, xmm2/m8           | A         | V/V                          | AVX2                     | Broadcast a byte integer in the source operand to sixteen locations in xmm1.                    |
| VEX.256.66.0F38.W0 78 /r<br>VPBROADCASTB ymm1, xmm2/m8           | A         | V/V                          | AVX2                     | Broadcast a byte integer in the source operand to thirty-two locations in ymm1.                 |
| EVEX.128.66.0F38.W0 78 /r<br>VPBROADCASTB xmm1{k1}{z}, xmm2/m8   | B         | V/V                          | AVX512VL<br>AVX512BW     | Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.    |
| EVEX.256.66.0F38.W0 78 /r<br>VPBROADCASTB ymm1{k1}{z}, xmm2/m8   | B         | V/V                          | AVX512VL<br>AVX512BW     | Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.    |
| EVEX.512.66.0F38.W0 78 /r<br>VPBROADCASTB zmm1{k1}{z}, xmm2/m8   | B         | V/V                          | AVX512BW                 | Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1. |
| VEX.128.66.0F38.W0 79 /r<br>VPBROADCASTW xmm1, xmm2/m16          | A         | V/V                          | AVX2                     | Broadcast a word integer in the source operand to eight locations in xmm1.                      |
| VEX.256.66.0F38.W0 79 /r<br>VPBROADCASTW ymm1, xmm2/m16          | A         | V/V                          | AVX2                     | Broadcast a word integer in the source operand to sixteen locations in ymm1.                    |
| EVEX.128.66.0F38.W0 79 /r<br>VPBROADCASTW xmm1{k1}{z}, xmm2/m16  | B         | V/V                          | AVX512VL<br>AVX512BW     | Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.    |
| EVEX.256.66.0F38.W0 79 /r<br>VPBROADCASTW ymm1{k1}{z}, xmm2/m16  | B         | V/V                          | AVX512VL<br>AVX512BW     | Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.    |
| EVEX.512.66.0F38.W0 79 /r<br>VPBROADCASTW zmm1{k1}{z}, xmm2/m16  | B         | V/V                          | AVX512BW                 | Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1. |
| VEX.128.66.0F38.W0 58 /r<br>VPBROADCASTD xmm1, xmm2/m32          | A         | V/V                          | AVX2                     | Broadcast a dword integer in the source operand to four locations in xmm1.                      |
| VEX.256.66.0F38.W0 58 /r<br>VPBROADCASTD ymm1, xmm2/m32          | A         | V/V                          | AVX2                     | Broadcast a dword integer in the source operand to eight locations in ymm1.                     |
| EVEX.128.66.0F38.W0 58 /r<br>VPBROADCASTD xmm1 {k1}{z}, xmm2/m32 | B         | V/V                          | AVX512VL<br>AVX512F      | Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.   |
| EVEX.256.66.0F38.W0 58 /r<br>VPBROADCASTD ymm1 {k1}{z}, xmm2/m32 | B         | V/V                          | AVX512VL<br>AVX512F      | Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.   |
| EVEX.512.66.0F38.W0 58 /r<br>VPBROADCASTD zmm1 {k1}{z}, xmm2/m32 | B         | V/V                          | AVX512F                  | Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.   |
| VEX.128.66.0F38.W0 59 /r<br>VPBROADCASTQ xmm1, xmm2/m64          | A         | V/V                          | AVX2                     | Broadcast a qword element in source operand to two locations in xmm1.                           |
| VEX.256.66.0F38.W0 59 /r<br>VPBROADCASTQ ymm1, xmm2/m64          | A         | V/V                          | AVX2                     | Broadcast a qword element in source operand to four locations in ymm1.                          |
| EVEX.128.66.0F38.W1 59 /r<br>VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64 | B         | V/V                          | AVX512VL<br>AVX512F      | Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.       |
| EVEX.256.66.0F38.W1 59 /r<br>VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64 | B         | V/V                          | AVX512VL<br>AVX512F      | Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.       |
| EVEX.512.66.0F38.W1 59 /r<br>VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64 | B         | V/V                          | AVX512F                  | Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.       |

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 59 /r<br>VBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64 | C          | V/V                          | AVX512VL<br>AVX512DQ     | Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.    |
| EVEX.256.66.0F38.W0 59 /r<br>VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64 | C          | V/V                          | AVX512VL<br>AVX512DQ     | Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.    |
| EVEX.512.66.0F38.W0 59 /r<br>VBROADCASTI32x2 zmm1 {k1}{z}, xmm2/m64 | C          | V/V                          | AVX512DQ                 | Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.    |
| VEX.256.66.0F38.W0 5A /r<br>VBROADCASTI128 ymm1, m128               | A          | V/V                          | AVX2                     | Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.                     |
| EVEX.256.66.0F38.W0 5A /r<br>VBROADCASTI32X4 ymm1 {k1}{z}, m128     | D          | V/V                          | AVX512VL<br>AVX512F      | Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 5A /r<br>VBROADCASTI32X4 zmm1 {k1}{z}, m128     | D          | V/V                          | AVX512F                  | Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 5A /r<br>VBROADCASTI64X2 ymm1 {k1}{z}, m128     | C          | V/V                          | AVX512VL<br>AVX512DQ     | Broadcast 128 bits of 2 quadword integer data in mem to locations in ymm1 using writemask k1.   |
| EVEX.512.66.0F38.W1 5A /r<br>VBROADCASTI64X2 zmm1 {k1}{z}, m128     | C          | V/V                          | AVX512DQ                 | Broadcast 128 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.   |
| EVEX.512.66.0F38.W0 5B /r<br>VBROADCASTI32X8 zmm1 {k1}{z}, m256     | E          | V/V                          | AVX512DQ                 | Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1. |
| EVEX.512.66.0F38.W1 5B /r<br>VBROADCASTI64X4 zmm1 {k1}{z}, m256     | D          | V/V                          | AVX512F                  | Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.   |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | N/A           | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| B     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| C     | Tuple2        | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| D     | Tuple4        | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| E     | Tuple8        | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

#### 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 ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VBROADCASTI32X4 and VBROADCASTI64X4 are reserved and will #UD.

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

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

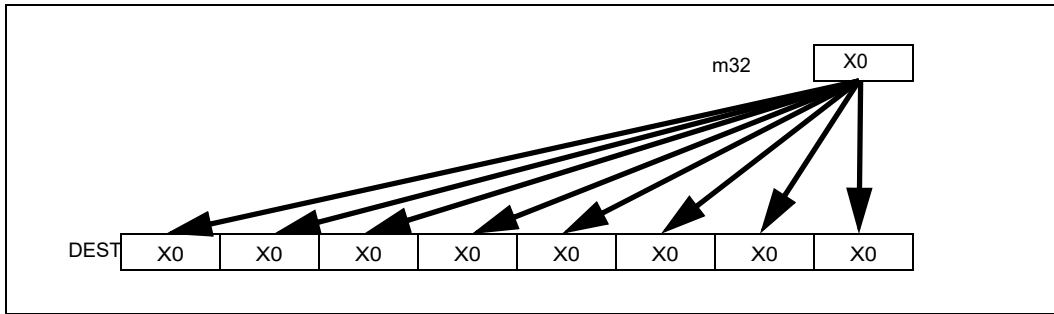


Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

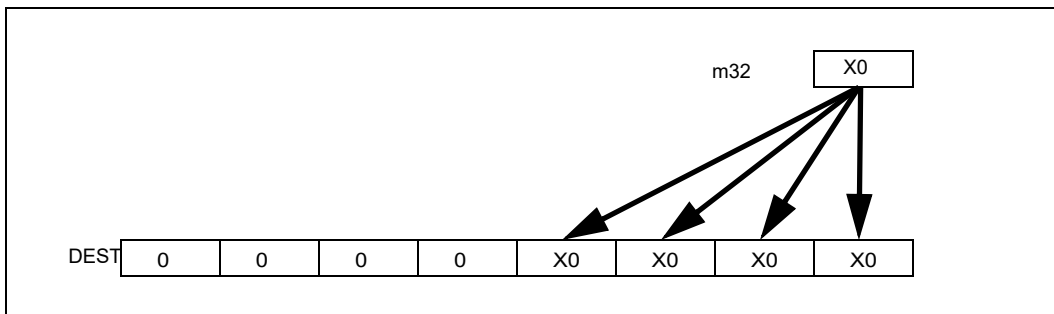


Figure 5-17. VPBROADCASTD Operation (128-bit version)

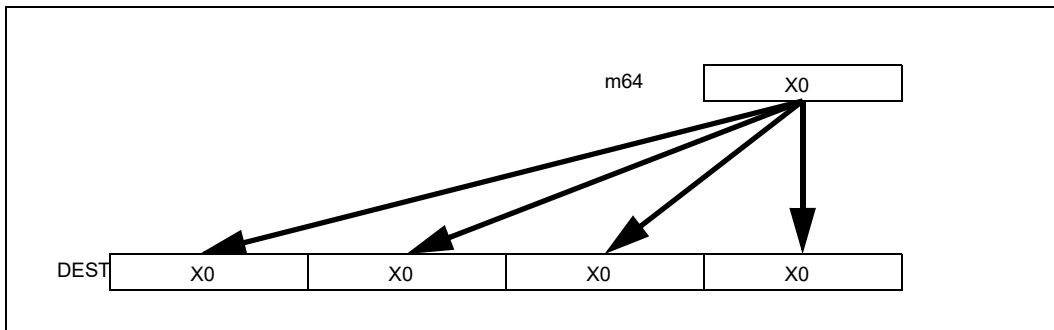


Figure 5-18. VPBROADCASTQ Operation (256-bit version)

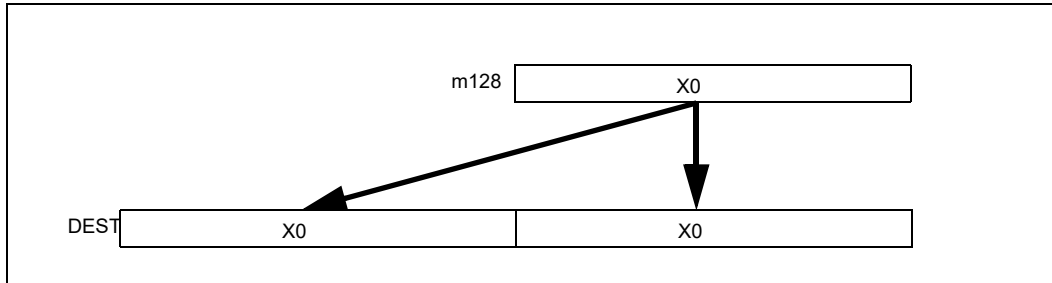


Figure 5-19. VBROADCASTI128 Operation (256-bit version)

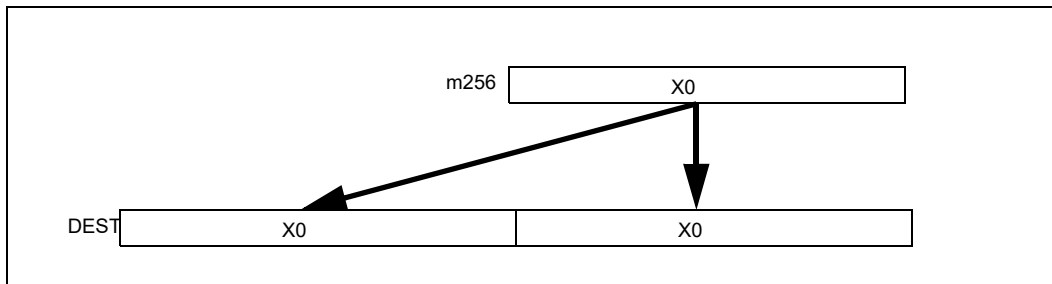


Figure 5-20. VBROADCASTI256 Operation (512-bit version)

**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\*

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 (128 bit version)**

temp := SRC[31:0]

DEST[31:0] := temp

DEST[63:32] := temp

DEST[95:64] := temp

DEST[127:96] := temp

DEST[MAXVL-1:128] := 0

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

temp := SRC[31:0]

DEST[31:0] := temp

DEST[63:32] := temp

DEST[95:64] := temp

DEST[127:96] := temp

DEST[159:128] := temp

DEST[191:160] := temp

DEST[223:192] := temp

DEST[255:224] := temp

DEST[MAXVL-1:256] := 0

**VPBROADCASTD (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 32

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

THEN DEST[i+31:i] := SRC[31:0]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0



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

```
temp := SRC[63:0]
DEST[63:0] := temp
DEST[127:64] := temp
DEST[191:128] := temp
DEST[255:192] := temp
DEST[MAXVL-1:256] := 0
```

**VPBROADCASTQ (EVEX encoded versions)**

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

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

**VBROADCASTI32x2 (EVEX encoded versions)**

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

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

**VBROADCASTI128 (VEX.256 encoded version)**

```
temp := SRC[127:0]
DEST[127:0] := temp
DEST[255:128] := temp
DEST[MAXVL-1:256] := 0
```

**VBROADCASTI32X4 (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 32

n := (j modulo 4) \* 32

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

THEN DEST[i+31:i] := SRC[n+31:n]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**VBROADCASTI64X2 (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 64

n := (j modulo 2) \* 64

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

THEN DEST[i+63:i] := SRC[n+63:n]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+63:i] = 0

FI

FI;

ENDFOR;

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

FOR j := 0 TO 15

i := j \* 32

n := (j modulo 8) \* 32

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

THEN DEST[i+31:i] := SRC[n+31:n]

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**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**

```

VPBROADCASTB __m512i __mm512_broadcastb_epi8( __m128i a);
VPBROADCASTB __m512i __mm512_mask_broadcastb_epi8(__m512i s, __mmask64 k, __m128i a);
VPBROADCASTB __m512i __mm512_maskz_broadcastb_epi8( __mmask64 k, __m128i a);
VPBROADCASTB __m256i __mm256_broadcastb_epi8(__m128i a);
VPBROADCASTB __m256i __mm256_mask_broadcastb_epi8(__m256i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i __mm256_maskz_broadcastb_epi8( __mmask32 k, __m128i a);
VPBROADCASTB __m128i __mm_mask_broadcastb_epi8(__m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i __mm_maskz_broadcastb_epi8( __mmask16 k, __m128i a);
VPBROADCASTB __m128i __mm_broadcastb_epi8(__m128i a);
VPBROADCASTD __m512i __mm512_broadcastd_epi32( __m128i a);
VPBROADCASTD __m512i __mm512_mask_broadcastd_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCASTD __m512i __mm512_maskz_broadcastd_epi32( __mmask16 k, __m128i a);
VPBROADCASTD __m256i __mm256_broadcastd_epi32( __m128i a);
VPBROADCASTD __m256i __mm256_mask_broadcastd_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTD __m256i __mm256_maskz_broadcastd_epi32( __mmask8 k, __m128i a);
VPBROADCASTD __m128i __mm_broadcastd_epi32(__m128i a);
VPBROADCASTD __m128i __mm_mask_broadcastd_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTD __m128i __mm_maskz_broadcastd_epi32( __mmask8 k, __m128i a);
VPBROADCASTQ __m512i __mm512_broadcastq_epi64( __m128i a);
VPBROADCASTQ __m512i __mm512_mask_broadcastq_epi64(__m512i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m512i __mm512_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTQ __m256i __mm256_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m256i __mm256_mask_broadcastq_epi64(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m256i __mm256_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTQ __m128i __mm_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m128i __mm_mask_broadcastq_epi64(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m128i __mm_maskz_broadcastq_epi64( __mmask8 k, __m128i a);
VPBROADCASTW __m512i __mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i __mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCASTW __m512i __mm512_maskz_broadcastw_epi16( __mmask32 k, __m128i a);
VPBROADCASTW __m256i __mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i __mm256_mask_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCASTW __m256i __mm256_maskz_broadcastw_epi16( __mmask16 k, __m128i a);
VPBROADCASTW __m128i __mm_broadcastw_epi16(__m128i a);
VPBROADCASTW __m128i __mm_mask_broadcastw_epi16(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTW __m128i __mm_maskz_broadcastw_epi16( __mmask8 k, __m128i a);
VBROADCASTI32x2 __m512i __mm512_broadcast_j32x2( __m128i a);

```

VBROADCASTI32x2 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x2(\_\_m512i s, \_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x2(\_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m256i \_\_mm256\_broadcast\_i32x2(\_\_m128i a);  
 VBROADCASTI32x2 \_\_m256i \_\_mm256\_mask\_broadcast\_i32x2(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m256i \_\_mm256\_maskz\_broadcast\_i32x2(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m128i \_\_mm\_broadcast\_i32x2(\_\_m128i a);  
 VBROADCASTI32x2 \_\_m128i \_\_mm\_mask\_broadcast\_i32x2(\_\_m128i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x2 \_\_m128i \_\_mm\_maskz\_broadcast\_i32x2(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m512i \_\_mm512\_broadcast\_i32x4(\_\_m128i a);  
 VBROADCASTI32x4 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x4(\_\_m512i s, \_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x4(\_\_mmask16 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_broadcast\_i32x4(\_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_mask\_broadcast\_i32x4(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x4 \_\_m256i \_\_mm256\_maskz\_broadcast\_i32x4(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_broadcast\_i32x8(\_\_m256i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_mask\_broadcast\_i32x8(\_\_m512i s, \_\_mmask16 k, \_\_m256i a);  
 VBROADCASTI32x8 \_\_m512i \_\_mm512\_maskz\_broadcast\_i32x8(\_\_mmask16 k, \_\_m256i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_broadcast\_i64x2(\_\_m128i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_mask\_broadcast\_i64x2(\_\_m512i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m512i \_\_mm512\_maskz\_broadcast\_i64x2(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_broadcast\_i64x2(\_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_mask\_broadcast\_i64x2(\_\_m256i s, \_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x2 \_\_m256i \_\_mm256\_maskz\_broadcast\_i64x2(\_\_mmask8 k, \_\_m128i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_broadcast\_i64x4(\_\_m256i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_mask\_broadcast\_i64x4(\_\_m512i s, \_\_mmask8 k, \_\_m256i a);  
 VBROADCASTI64x4 \_\_m512i \_\_mm512\_maskz\_broadcast\_i64x4(\_\_mmask8 k, \_\_m256i a);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

EVEX-encoded instructions, see Table 2-23, “Type 6 Class Exception Conditions.”

EVEX-encoded instructions, syntax with reg/mem operand, see Table 2-53, “Type E6 Class Exception Conditions.”

Additionally:

#UD                    If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.  
                           If EVEX.L'L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.  
                           If EVEX.L'L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.

## VPBROADCASTM—Broadcast Mask to Vector Register

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

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

#### Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128).

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

#### Operation

##### VPBROADCASTMB2Q

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

FOR j := 0 TO KL-1

  i := j\*64

  DEST[i+63:i] := ZeroExtend(SRC[7:0])

ENDFOR

DEST[MAXVL-1:VL] := 0

##### VPBROADCASTMW2D

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

FOR j := 0 TO KL-1

  i := j\*32

  DEST[i+31:i] := ZeroExtend(SRC[15:0])

ENDFOR

DEST[MAXVL-1:VL] := 0

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

VPBROADCASTMB2Q \_\_m512i \_\_mm512\_broadcastmb\_epi64(\_\_mmask8);  
VPBROADCASTMW2D \_\_m512i \_\_mm512\_broadcastmw\_epi32(\_\_mmask16);  
VPBROADCASTMB2Q \_\_m256i \_\_mm256\_broadcastmb\_epi64(\_\_mmask8);  
VPBROADCASTMW2D \_\_m256i \_\_mm256\_broadcastmw\_epi32(\_\_mmask8);  
VPBROADCASTMB2Q \_\_m128i \_\_mm\_broadcastmb\_epi64(\_\_mmask8);  
VPBROADCASTMW2D \_\_m128i \_\_mm\_broadcastmw\_epi32(\_\_mmask8);

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Table 2-54, “Type E6NF Class Exception Conditions.”

## VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W0 3F /r ib<br>VPCMPB k1 {k2}, xmm2,<br>xmm3/m128, imm8  | A         | V/V                         | AVX512VL<br>AVX512BW     | 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.   |
| EVEX.256.66.0F3A.W0 3F /r ib<br>VPCMPB k1 {k2}, ymm2,<br>ymm3/m256, imm8  | A         | V/V                         | AVX512VL<br>AVX512BW     | 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.   |
| EVEX.512.66.0F3A.W0 3F /r ib<br>VPCMPB k1 {k2}, zmm2,<br>zmm3/m512, imm8  | A         | V/V                         | AVX512BW                 | 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.   |
| EVEX.128.66.0F3A.W0 3E /r ib<br>VPCMPUB k1 {k2}, xmm2,<br>xmm3/m128, imm8 | A         | V/V                         | AVX512VL<br>AVX512BW     | 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. |
| EVEX.256.66.0F3A.W0 3E /r ib<br>VPCMPUB k1 {k2}, ymm2,<br>ymm3/m256, imm8 | A         | V/V                         | AVX512VL<br>AVX512BW     | 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. |
| EVEX.512.66.0F3A.W0 3E /r ib<br>VPCMPUB k1 {k2}, zmm2,<br>zmm3/m512, imm8 | A         | V/V                         | AVX512BW                 | 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. |

### Instruction Operand Encoding

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

### 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 5-11.

**Table 5-11. Pseudo-Op and VPCMP\* Implementation**

| Pseudo-Op                          | PCMPM Implementation              |
|------------------------------------|-----------------------------------|
| VPCMPEQ* <i>reg1, reg2, reg3</i>   | VPCMP* <i>reg1, reg2, reg3, 0</i> |
| VPCMPLT* <i>reg1, reg2, reg3</i>   | VPCMP* <i>reg1, reg2, reg3, 1</i> |
| VPCMPLE* <i>reg1, reg2, reg3</i>   | VPCMP* <i>reg1, reg2, reg3, 2</i> |
| VPCMPNEQ* <i>reg1, reg2, reg3</i>  | VPCMP* <i>reg1, reg2, reg3, 4</i> |
| VPPCMPNLT* <i>reg1, reg2, reg3</i> | VPCMP* <i>reg1, reg2, reg3, 5</i> |
| VPCMPNLE* <i>reg1, reg2, reg3</i>  | VPCMP* <i>reg1, reg2, reg3, 6</i> |

### 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[j+7:i] OP SRC2[j+7:i];

      IF CMP = TRUE

        THEN DEST[j] := 1;

        ELSE DEST[j] := 0; FI;

    ELSE DEST[j] = 0 ; zeroing-masking onlyFI;

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0



**VPCMPUB (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 8

IF k2[j] OR \*no writemask\*

THEN

CMP := SRC1[i+7:i] OP SRC2[i+7:i];

IF CMP = TRUE

THEN DEST[j] := 1;

ELSE DEST[j] := 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

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

VPCMPB \_\_mmask64 \_\_mm512\_cmp\_epi8\_mask( \_\_m512i a, \_\_m512i b, int cmp);

VPCMPB \_\_mmask64 \_\_mm512\_mask\_cmp\_epi8\_mask( \_\_mmask64 m, \_\_m512i a, \_\_m512i b, int cmp);

VPCMPB \_\_mmask32 \_\_mm256\_cmp\_epi8\_mask( \_\_m256i a, \_\_m256i b, int cmp);

VPCMPB \_\_mmask32 \_\_mm256\_mask\_cmp\_epi8\_mask( \_\_mmask32 m, \_\_m256i a, \_\_m256i b, int cmp);

VPCMPB \_\_mmask16 \_\_mm128\_cmp\_epi8\_mask( \_\_m128i a, \_\_m128i b, int cmp);

VPCMPB \_\_mmask16 \_\_mm128\_mask\_cmp\_epi8\_mask( \_\_mmask16 m, \_\_m128i a, \_\_m128i b, int cmp);

VPCMPB \_\_mmask64 \_\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_m512i a, \_\_m512i b);

VPCMPB \_\_mmask64 \_\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_mmask64 m, \_\_m512i a, \_\_m512i b);

VPCMPB \_\_mmask32 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_m256i a, \_\_m256i b);

VPCMPB \_\_mmask32 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_mmask32 m, \_\_m256i a, \_\_m256i b);

VPCMPB \_\_mmask16 \_\_mm128\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_m128i a, \_\_m128i b);

VPCMPB \_\_mmask16 \_\_mm128\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_mmask16 m, \_\_m128i a, \_\_m128i b);

VPCMPUB \_\_mmask64 \_\_mm512\_cmp\_epu8\_mask( \_\_m512i a, \_\_m512i b, int cmp);

VPCMPUB \_\_mmask64 \_\_mm512\_mask\_cmp\_epu8\_mask( \_\_mmask64 m, \_\_m512i a, \_\_m512i b, int cmp);

VPCMPUB \_\_mmask32 \_\_mm256\_cmp\_epu8\_mask( \_\_m256i a, \_\_m256i b, int cmp);

VPCMPUB \_\_mmask32 \_\_mm256\_mask\_cmp\_epu8\_mask( \_\_mmask32 m, \_\_m256i a, \_\_m256i b, int cmp);

VPCMPUB \_\_mmask16 \_\_mm128\_cmp\_epu8\_mask( \_\_m128i a, \_\_m128i b, int cmp);

VPCMPUB \_\_mmask16 \_\_mm128\_mask\_cmp\_epu8\_mask( \_\_mmask16 m, \_\_m128i a, \_\_m128i b, int cmp);

VPCMPUB \_\_mmask64 \_\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epu8\_mask( \_\_m512i a, \_\_m512i b, int cmp);

VPCMPUB \_\_mmask64 \_\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu8\_mask( \_\_mmask64 m, \_\_m512i a, \_\_m512i b, int cmp);

VPCMPUB \_\_mmask32 \_\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epu8\_mask( \_\_m256i a, \_\_m256i b, int cmp);

VPCMPUB \_\_mmask32 \_\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu8\_mask( \_\_mmask32 m, \_\_m256i a, \_\_m256i b, int cmp);

VPCMPUB \_\_mmask16 \_\_mm128\_cmp[eq|ge|gt|le|lt|neq]\_epu8\_mask( \_\_m128i a, \_\_m128i b, int cmp);

VPCMPUB \_\_mmask16 \_\_mm128\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epu8\_mask( \_\_mmask16 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."

## VPCMPD/VPCMPUD—Compare Packed Integer Values Into Mask

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W0 1F /r ib<br>VPCMPD k1 {k2}, xmm2,<br>xmm3/m128/m32bcst, imm8  | A         | V/V                         | AVX512VL<br>AVX512F      | Compare packed signed doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.                     |
| EVEX.256.66.0F3A.W0 1F /r ib<br>VPCMPD k1 {k2}, ymm2,<br>ymm3/m256/m32bcst, imm8  | A         | V/V                         | AVX512VL<br>AVX512F      | Compare packed signed doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.                     |
| EVEX.512.66.0F3A.W0 1F /r ib<br>VPCMPD k1 {k2}, zmm2,<br>zmm3/m512/m32bcst, imm8  | A         | V/V                         | AVX512F                  | Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.   |
| EVEX.128.66.0F3A.W0 1E /r ib<br>VPCMPUD k1 {k2}, xmm2,<br>xmm3/m128/m32bcst, imm8 | A         | V/V                         | AVX512VL<br>AVX512F      | Compare packed unsigned doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.                   |
| EVEX.256.66.0F3A.W0 1E /r ib<br>VPCMPUD k1 {k2}, ymm2,<br>ymm3/m256/m32bcst, imm8 | A         | V/V                         | AVX512VL<br>AVX512F      | Compare packed unsigned doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.                   |
| EVEX.512.66.0F3A.W0 1E /r ib<br>VPCMPUD k1 {k2}, zmm2,<br>zmm3/m512/m32bcst, imm8 | A         | V/V                         | AVX512F                  | Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2. |

### Instruction Operand Encoding

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

### 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 5-11.

**Operation**

CASE (COMPARISON PREDICATE) OF

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

ESAC;

**VPCMPD (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 32

IF k2[j] OR \*no writemask\*

THEN

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

THEN CMP := SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] := 1;

ELSE DEST[j] := 0; FI;

ELSE DEST[j] := 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

**VPCMPUD (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 32

IF k2[j] OR \*no writemask\*

THEN

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

THEN CMP := SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP := SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] := 1;

ELSE DEST[j] := 0; FI;

ELSE DEST[j] := 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

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

```

VPCMPD __mmask16 _mm512_cmp_epi32_mask( __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m512i a, __m512i b);
VPCMPD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_cmp_epu32_mask( __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_mask_cmp_epu32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPD __mmask8 _mm256_cmp_epi32_mask( __m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m256i a, __m256i b);
VPCMPD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_cmp_epu32_mask( __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_mask_cmp_epu32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 _mm_cmp_epi32_mask( __m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m128i a, __m128i b);
VPCMPD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_cmp_epu32_mask( __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_mask_cmp_epu32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu32_mask( __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

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

## VPCMPQ/VPCMPUQ—Compare Packed Integer Values Into Mask

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W1 1F /r ib<br>VPCMPQ k1 {k2}, xmm2,<br>xmm3/m128/m64bcst, imm8  | A         | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.256.66.0F3A.W1 1F /r ib<br>VPCMPQ k1 {k2}, ymm2,<br>ymm3/m256/m64bcst, imm8  | A         | V/V                          | AVX512VL<br>AVX512F      | Compare packed signed quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.512.66.0F3A.W1 1F /r ib<br>VPCMPQ k1 {k2}, zmm2,<br>zmm3/m512/m64bcst, imm8  | A         | V/V                          | AVX512F                  | Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.   |
| EVEX.128.66.0F3A.W1 1E /r ib<br>VPCMPUQ k1 {k2}, xmm2,<br>xmm3/m128/m64bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512F      | Compare packed unsigned quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.256.66.0F3A.W1 1E /r ib<br>VPCMPUQ k1 {k2}, ymm2,<br>ymm3/m256/m64bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512F      | Compare packed unsigned quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |
| EVEX.512.66.0F3A.W1 1E /r ib<br>VPCMPUQ k1 {k2}, zmm2,<br>zmm3/m512/m64bcst, imm8 | A         | V/V                          | AVX512F                  | Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1. |

### Instruction Operand Encoding

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

### 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 5-11.

**Operation**

CASE (COMPARISON PREDICATE) OF

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

ESAC;

**VPCMPQ (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 64

IF k2[j] OR \*no writemask\*

THEN

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

THEN CMP := SRC1[i+63:i] OP SRC2[63:0];

ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] := 1;

ELSE DEST[j] := 0; FI;

ELSE DEST[j] := 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

**VPCMPUQ (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 64

IF k2[j] OR \*no writemask\*

THEN

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

THEN CMP := SRC1[i+63:i] OP SRC2[63:0];

ELSE CMP := SRC1[i+63:i] OP SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] := 1;

ELSE DEST[j] := 0; FI;

ELSE DEST[j] := 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

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

```

VPCMPQ __mmask8 _mm512_cmp_epi64_mask( __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m512i a, __m512i b);
VPCMPQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_cmp_epu64_mask( __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_mask_cmp_epu64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPQ __mmask8 _mm256_cmp_epi64_mask( __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m256i a, __m256i b);
VPCMPQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_cmp_epu64_mask( __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_mask_cmp_epu64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPQ __mmask8 _mm_cmp_epi64_mask( __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi64_mask( __m128i a, __m128i b);
VPCMPQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_cmp_epu64_mask( __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_mask_cmp_epu64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu64_mask( __m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

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

## VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W1 3F /r ib<br>VPCMPW k1 {k2}, xmm2,<br>xmm3/m128, imm8  | A         | V/V                          | AVX512VL<br>AVX512BW     | 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.   |
| EVEX.256.66.0F3A.W1 3F /r ib<br>VPCMPW k1 {k2}, ymm2,<br>ymm3/m256, imm8  | A         | V/V                          | AVX512VL<br>AVX512BW     | 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.   |
| EVEX.512.66.0F3A.W1 3F /r ib<br>VPCMPW k1 {k2}, zmm2,<br>zmm3/m512, imm8  | A         | V/V                          | AVX512BW                 | 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.   |
| EVEX.128.66.0F3A.W1 3E /r ib<br>VPCMPUW k1 {k2}, xmm2,<br>xmm3/m128, imm8 | A         | V/V                          | AVX512VL<br>AVX512BW     | 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. |
| EVEX.256.66.0F3A.W1 3E /r ib<br>VPCMPUW k1 {k2}, ymm2,<br>ymm3/m256, imm8 | A         | V/V                          | AVX512VL<br>AVX512BW     | 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. |
| EVEX.512.66.0F3A.W1 3E /r ib<br>VPCMPUW k1 {k2}, zmm2,<br>zmm3/m512, imm8 | A         | V/V                          | AVX512BW                 | 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. |

### Instruction Operand Encoding

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

### 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 5-11.



**Operation**

CASE (COMPARISON PREDICATE) OF

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

ESAC;

**VPCMPW (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 16

IF k2[j] OR \*no writemask\*

THEN

ICMP := SRC1[i+15:i] OP SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j] := 1;

ELSE DEST[j] := 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

**VPCMPUW (EVEX encoded versions)**

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

FOR j := 0 TO KL-1

i := j \* 16

IF k2[j] OR \*no writemask\*

THEN

CMP := SRC1[i+15:i] OP SRC2[i+15:i];

IF CMP = TRUE

THEN DEST[j] := 1;

ELSE DEST[j] := 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

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

```

VPCMPW __mmask32 __mm512_cmp_epi16_mask( __m512i a, __m512i b, int cmp);
VPCMPW __mmask32 __mm512_mask_cmp_epi16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPW __mmask16 __mm256_cmp_epi16_mask( __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_epu16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 __mm512_mask_cmp_epu16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 __mm256_cmp_epu16_mask( __m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 __mm256_mask_cmp_epu16_mask( __mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 __mm_cmp_epu16_mask( __m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 __mm_mask_cmp_epu16_mask( __mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPUW __mmask32 __mm512_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 __mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 __mm256_cmp[eq|ge|gt|le|lt|neq]_epu16_mask( __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

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

### Instruction Operand Encoding

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

### Description

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

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

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

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

This instruction supports memory fault suppression.

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

### Operation

#### VPCOMPRESSB store form

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

k := 0

FOR j := 0 TO KL-1:

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

DEST.byte[k] := SRC.byte[j]

k := k + 1

#### VPCOMPRESSB reg-reg form

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

k := 0

FOR j := 0 TO KL-1:

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

DEST.byte[k] := SRC.byte[j]

k := k + 1

IF \*merging-masking\*:

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

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

DEST[MAX\_VL-1:VL] := 0

#### VPCOMPRESSW store form

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

k := 0

FOR j := 0 TO KL-1:

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

DEST.word[k] := SRC.word[j]

k := k + 1

#### VPCOMPRESSW reg-reg form

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

k := 0

FOR j := 0 TO KL-1:

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

DEST.word[k] := SRC.word[j]

k := k + 1

IF \*merging-masking\*:

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

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

DEST[MAX\_VL-1:VL] := 0

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

```

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

```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

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

## VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values Into Dense Memory/Register

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 8B /r<br>VPCOMPRESSD xmm1/m128 {k1}{z}, xmm2 | A         | V/V                          | AVX512VL<br>AVX512F      | Compress packed doubleword integer values from xmm2 to xmm1/m128 using control mask k1. |
| EVEX.256.66.0F38.W0 8B /r<br>VPCOMPRESSD ymm1/m256 {k1}{z}, ymm2 | A         | V/V                          | AVX512VL<br>AVX512F      | Compress packed doubleword integer values from ymm2 to ymm1/m256 using control mask k1. |
| EVEX.512.66.0F38.W0 8B /r<br>VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2 | A         | V/V                          | AVX512F                  | Compress packed doubleword integer values from zmm2 to zmm1/m512 using control mask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

### 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."

## VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values Into Dense Memory/Register

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

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

### 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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 C4 /r<br>VPCONFLICTD xmm1 {k1}{z},<br>xmm2/m128/m32bcst | A         | V/V                          | AVX512VL<br>AVX512CD     | Detect duplicate double-word values in<br>xmm2/m128/m32bcst using writemask k1. |
| EVEX.256.66.0F38.W0 C4 /r<br>VPCONFLICTD ymm1 {k1}{z},<br>ymm2/m256/m32bcst | A         | V/V                          | AVX512VL<br>AVX512CD     | Detect duplicate double-word values in<br>ymm2/m256/m32bcst using writemask k1. |
| EVEX.512.66.0F38.W0 C4 /r<br>VPCONFLICTD zmm1 {k1}{z},<br>zmm2/m512/m32bcst | A         | V/V                          | AVX512CD                 | Detect duplicate double-word values in<br>zmm2/m512/m32bcst using writemask k1. |
| EVEX.128.66.0F38.W1 C4 /r<br>VPCONFLICTQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst | A         | V/V                          | AVX512VL<br>AVX512CD     | Detect duplicate quad-word values in<br>xmm2/m128/m64bcst using writemask k1.   |
| EVEX.256.66.0F38.W1 C4 /r<br>VPCONFLICTQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst | A         | V/V                          | AVX512VL<br>AVX512CD     | Detect duplicate quad-word values in<br>ymm2/m256/m64bcst using writemask k1.   |
| EVEX.512.66.0F38.W1 C4 /r<br>VPCONFLICTQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst | A         | V/V                          | AVX512CD                 | Detect duplicate quad-word values in<br>zmm2/m512/m64bcst using writemask k1.   |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element's comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

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

**Operation****VPCONFLICTD**

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

FOR j := 0 TO KL-1

i := j\*32

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

FOR k := 0 TO j-1

m := k\*32

IF ((SRC[i+31:i] = SRC[m+31:m])) THEN

DEST[i+k] := 1

ELSE

DEST[i+k] := 0

FI

ENDFOR

DEST[i+31:i+j] := 0

ELSE

IF \*merging-masking\* THEN

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

ELSE

DEST[i+31:i] := 0

FI

FI

ENDFOR

DEST[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

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

### Instruction Operand Encoding

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

### Description

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

This instruction supports memory fault suppression.

### Operation

**VPDPBUSD dest, src1, src2 (VEX encoded versions)**

VL=(128, 256)

KL=VL/32

ORIGDEST := DEST

FOR i := 0 TO KL-1:

    // Extending to 16b

    // src1extend := ZERO\_EXTEND

    // src2extend := SIGN\_EXTEND

    p1word := src1extend(SRC1.byte[4\*i+0]) \* src2extend(SRC2.byte[4\*i+0])

```

p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

```

```
DEST[MAX_VL-1:VL] := 0
```

#### VPDPBUSD dest, src1, src2 (EVEX encoded versions)

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

```
ORIGDEST := DEST
```

```
FOR i := 0 TO KL-1:
```

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

```
    // Byte elements of SRC1 are zero-extended to 16b and
```

```
    // byte elements of SRC2 are sign extended to 16b before multiplication.
```

```
    IF SRC2 is memory and EVEX.b == 1:
```

```
      t := SRC2.dword[0]
```

```
    ELSE:
```

```
      t := SRC2.dword[i]
```

```
    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);
```

```
VPDPBUSD __m256i __mm256_dpbusd_epi32(__m256i, __m256i, __m256i);
```

```
VPDPBUSD __m256i __mm256_mask_dpbusd_epi32(__m256i, __mmask8, __m256i, __m256i);
```

```
VPDPBUSD __m256i __mm256_maskz_dpbusd_epi32(__mmask8, __m256i, __m256i, __m256i);
```

```
VPDPBUSD __m512i __mm512_dpbusd_epi32(__m512i, __m512i, __m512i);
```

```
VPDPBUSD __m512i __mm512_mask_dpbusd_epi32(__m512i, __mmask16, __m512i, __m512i);
```

```
VPDPBUSD __m512i __mm512_maskz_dpbusd_epi32(__mmask16, __m512i, __m512i, __m512i);
```

#### SIMD Floating-Point Exceptions

None.

#### Other Exceptions

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

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

### Instruction Operand Encoding

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

#### 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 or 0x8000\_0000 for negative numbers.

This instruction supports memory fault suppression.

#### Operation

**VPDPBUSDS dest, src1, src2 (VEX encoded versions)**

VL=(128, 256)

KL=VL/32

ORIGDEST := DEST

FOR i := 0 TO KL-1:

```

// Extending to 16b
// src1extend := ZERO_EXTEND
// src2extend := SIGN_EXTEND

p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])
DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)

DEST[MAX_VL-1:VL] := 0

VPDPBUSDS dest, src1, src2 (EVEX encoded versions)
(KL,VL)=(4,128), (8,256), (16,512)
ORIGDEST := DEST
FOR i := 0 TO KL-1:
  IF k1[i] or *no writemask*:
    // Byte elements of SRC1 are zero-extended to 16b and
    // byte elements of SRC2 are sign extended to 16b before multiplication.
    IF SRC2 is memory and EVEX.b == 1:
      t := SRC2.dword[0]
    ELSE:
      t := SRC2.dword[i]
    p1word := ZERO_EXTEND(SRC1.byte[4*i]) * SIGN_EXTEND(t.byte[0])
    p2word := ZERO_EXTEND(SRC1.byte[4*i+1]) * SIGN_EXTEND(t.byte[1])
    p3word := ZERO_EXTEND(SRC1.byte[4*i+2]) * SIGN_EXTEND(t.byte[2])
    p4word := ZERO_EXTEND(SRC1.byte[4*i+3]) * SIGN_EXTEND(t.byte[3])
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)
  ELSE IF *zeroing*:
    DEST.dword[i] := 0
  ELSE: // Merge masking, dest element unchanged
    DEST.dword[i] := ORIGDEST.dword[i]
DEST[MAX_VL-1:VL] := 0

```

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

```

VPDPBUSDS __m128i _mm_dpbusds_avx_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_dpbusds_epi32(__m128i, __m128i, __m128i);
VPDPBUSDS __m128i _mm_mask_dpbusds_epi32(__m128i, __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(__m256i, __mmask8, __m256i, __m256i);
VPDPBUSDS __m256i _mm256_maskz_dpbusds_epi32(__mmask8, __m256i, __m256i, __m256i);
VPDPBUSDS __m512i _mm512_dpbusds_epi32(__m512i, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_mask_dpbusds_epi32(__m512i, __mmask16, __m512i, __m512i);
VPDPBUSDS __m512i _mm512_maskz_dpbusds_epi32(__mmask16, __m512i, __m512i, __m512i);

```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

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

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

### Instruction Operand Encoding

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

### 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

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

### Instruction Operand Encoding

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

#### 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 or 0x8000\_0000 for negative numbers.

This instruction supports memory fault suppression.

**Operation****VPDPWSSDS dest, src1, src2 (VEX encoded versions)**

VL=(128, 256)

KL=VL/32

ORIGDEST := DEST

FOR i := 0 TO KL-1:

p1dword := SIGN\_EXTEND(SRC1.word[2\*i+0]) \* SIGN\_EXTEND(SRC2.word[2\*i+0])

p2dword := SIGN\_EXTEND(SRC1.word[2\*i+1]) \* SIGN\_EXTEND(SRC2.word[2\*i+1])

DEST.dword[i] := SIGNED\_DWORD\_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)

DEST[MAX\_VL-1:VL] := 0

**VPDPWSSDS dest, src1, src2 (EVEX encoded versions)**

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

ORIGDEST := DEST

FOR i := 0 TO KL-1:

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

IF SRC2 is memory and EVEX.b == 1:

t := SRC2.dword[0]

ELSE:

t := SRC2.dword[i]

p1dword := SIGN\_EXTEND(SRC1.word[2\*i]) \* SIGN\_EXTEND(t.word[0])

p2dword := SIGN\_EXTEND(SRC1.word[2\*i+1]) \* SIGN\_EXTEND(t.word[1])

DEST.dword[i] := SIGNED\_DWORD\_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)

ELSE IF \*zeroing\*:

DEST.dword[i] := 0

ELSE: // Merge masking, dest element unchanged

DEST.dword[i] := ORIGDEST.dword[i]

DEST[MAX\_VL-1:VL] := 0

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

VPDPWSSDS \_\_m128i \_\_mm\_dpwssds\_avx\_epi32(\_\_m128i, \_\_m128i, \_\_m128i);

VPDPWSSDS \_\_m128i \_\_mm\_dpwssds\_epi32(\_\_m128i, \_\_m128i, \_\_m128i);

VPDPWSSDS \_\_m128i \_\_mm\_mask\_dpwssd\_epi32(\_\_m128i, \_\_mmask8, \_\_m128i, \_\_m128i);

VPDPWSSDS \_\_m128i \_\_mm\_maskz\_dpwssd\_epi32(\_\_mmask8, \_\_m128i, \_\_m128i, \_\_m128i);

VPDPWSSDS \_\_m256i \_\_mm256\_dpwssds\_avx\_epi32(\_\_m256i, \_\_m256i, \_\_m256i);

VPDPWSSDS \_\_m256i \_\_mm256\_dpwssd\_epi32(\_\_m256i, \_\_m256i, \_\_m256i);

VPDPWSSDS \_\_m256i \_\_mm256\_mask\_dpwssd\_epi32(\_\_m256i, \_\_mmask8, \_\_m256i, \_\_m256i);

VPDPWSSDS \_\_m256i \_\_mm256\_maskz\_dpwssd\_epi32(\_\_mmask8, \_\_m256i, \_\_m256i, \_\_m256i);

VPDPWSSDS \_\_m512i \_\_mm512\_dpwssd\_epi32(\_\_m512i, \_\_m512i, \_\_m512i);

VPDPWSSDS \_\_m512i \_\_mm512\_mask\_dpwssd\_epi32(\_\_m512i, \_\_mmask16, \_\_m512i, \_\_m512i);

VPDPWSSDS \_\_m512i \_\_mm512\_maskz\_dpwssd\_epi32(\_\_mmask16, \_\_m512i, \_\_m512i, \_\_m512i);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, "Type 4 Class Exception Conditions."

EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions."

## VPERM2F128—Permute Floating-Point Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.256.66.0F3A.W0 06 /r ib<br>VPERM2F128 ymm1, ymm2, ymm3/m256, imm8 | RV<br>MI  | V/V                          | AVX                      | Permute 128-bit floating-point fields in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|---------------|--------------|---------------|-----------|
| RVMI  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### Description

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

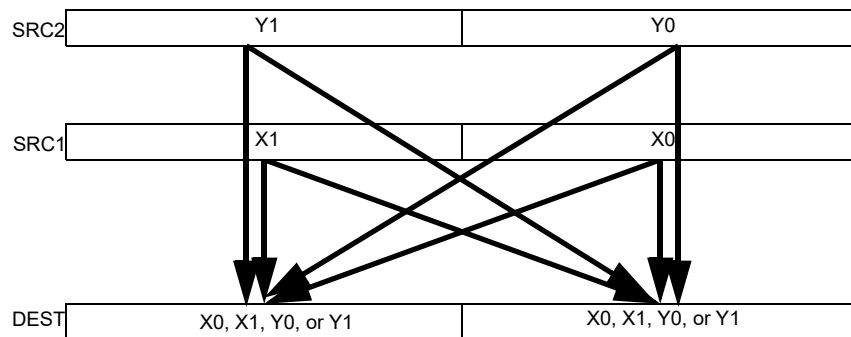


Figure 5-21. VPERM2F128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.

## Operation

### VPERM2F128

CASE IMM8[1:0] of

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

1: DEST[127:0] := SRC1[255:128]

2: DEST[127:0] := SRC2[127:0]

3: DEST[127:0] := SRC2[255:128]

ESAC

CASE IMM8[5:4] of

0: DEST[255:128] := SRC1[127:0]

1: DEST[255:128] := SRC1[255:128]

2: DEST[255:128] := SRC2[127:0]

3: DEST[255:128] := SRC2[255:128]

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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>-bit<br>Mode | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------|--------------------------|--|
| VEX.256.66.0F3A.W0 46 /r ib<br>VPERM2I128 ymm1, ymm2, ymm3/m256, imm8 | RVMI      | V/V                   | AVX2                     | Permute 128-bit integer data in ymm2 and ymm3/mem using controls from imm8 and store result in ymm1. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|---------------|--------------|---------------|-----------|
| RVMI  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### Description

Permute 128 bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

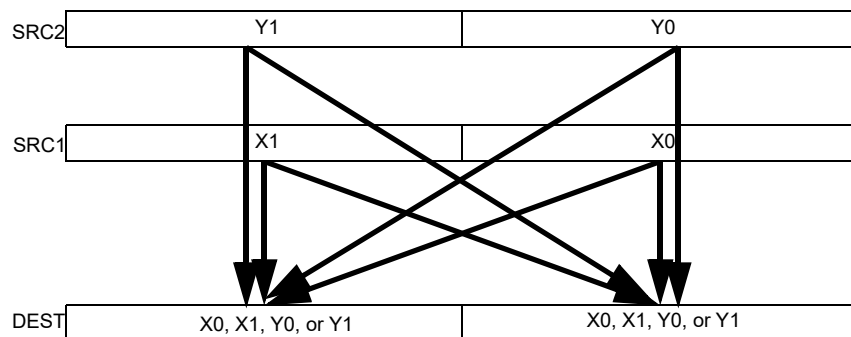


Figure 5-22. VPERM2I128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed. VEX.L must be 1, otherwise the instruction will #UD.

**Operation****VPERM2I128**

CASE IMM8[1:0] of

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

1: DEST[127:0] := SRC1[255:128]

2: DEST[127:0] := SRC2[127:0]

3: DEST[127:0] := SRC2[255:128]

ESAC

CASE IMM8[5:4] of

0: DEST[255:128] := SRC1[127:0]

1: DEST[255:128] := SRC1[255:128]

2: DEST[255:128] := SRC2[127:0]

3: DEST[255:128] := SRC2[255:128]

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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|-----------|------------------------------|-------------------------|--|
| EVEX.128.66.0F38.W0 8D /r<br>VPERMB xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | A         | V/V                          | AVX512VL<br>AVX512_VBMI | Permute bytes in xmm3/m128 using byte indexes in xmm2 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 8D /r<br>VPERMB ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | A         | V/V                          | AVX512VL<br>AVX512_VBMI | Permute bytes in ymm3/m256 using byte indexes in ymm2 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 8D /r<br>VPERMB zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | A         | V/V                          | AVX512_VBMI             | Permute bytes in zmm3/m512 using byte indexes in zmm2 and store the result in zmm1 using writemask k1. |

### Instruction Operand Encoding

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

### Description

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

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

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

### Operation

#### VPERMB (EVEX encoded versions)

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

IF VL = 128:

n := 3;

ELSE IF VL = 256:

n := 4;

ELSE IF VL = 512:

n := 5;

FI;

FOR j := 0 TO KL-1:

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

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

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

ELSE IF zeroing-masking THEN

DEST[j\*8 + 7 : j\*8] := 0;

\*ELSE

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

FI

ENDFOR

DEST[MAX\_VL-1:VL] := 0;

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

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

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."

## VPERMD/VPERMW—Permute Packed Doubleword/Word Elements

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| VEX.256.66.0F38.W0 36 /r<br>VPERMD ymm1, ymm2, ymm3/m256                     | A          | V/V                          | AVX2                     | Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.                            |
| EVEX.256.66.0F38.W0 36 /r<br>VPERMD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Permute doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 36 /r<br>VPERMD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | B          | V/V                          | AVX512F                  | Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1. |
| EVEX.128.66.0F38.W1 8D /r<br>VPERMW xmm1 {k1}{z}, xmm2,<br>xmm3/m128         | C          | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.       |
| EVEX.256.66.0F38.W1 8D /r<br>VPERMW ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | C          | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers in ymm3/m256 using indexes in ymm2 and store the result in ymm1 using writemask k1.       |
| EVEX.512.66.0F38.W1 8D /r<br>VPERMW zmm1 {k1}{z}, zmm2,<br>zmm3/m512         | C          | V/V                          | AVX512BW                 | Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.       |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| C     | Full Mem   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

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

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

**Operation****VPERMD (EVEX encoded versions)**

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

IF VL = 256 THEN n := 2; FI;

IF VL = 512 THEN n := 3; FI;

FOR j := 0 TO KL-1

i := j \* 32

id := 32\*SRC1[i+n:i]

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

THEN

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

THEN DEST[i+31:i] := SRC2[31:0];

ELSE DEST[i+31:i] := SRC2[id+31:id];

FI;

ELSE

IF \*merging-masking\* ; merging-masking

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

ELSE ; zeroing-masking

DEST[i+31:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

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

DEST[31:0] := (SRC2[255:0] &gt;&gt; (SRC1[2:0] \* 32))[31:0];

DEST[63:32] := (SRC2[255:0] &gt;&gt; (SRC1[34:32] \* 32))[31:0];

DEST[95:64] := (SRC2[255:0] &gt;&gt; (SRC1[66:64] \* 32))[31:0];

DEST[127:96] := (SRC2[255:0] &gt;&gt; (SRC1[98:96] \* 32))[31:0];

DEST[159:128] := (SRC2[255:0] &gt;&gt; (SRC1[130:128] \* 32))[31:0];

DEST[191:160] := (SRC2[255:0] &gt;&gt; (SRC1[162:160] \* 32))[31:0];

DEST[223:192] := (SRC2[255:0] &gt;&gt; (SRC1[194:192] \* 32))[31:0];

DEST[255:224] := (SRC2[255:0] &gt;&gt; (SRC1[226:224] \* 32))[31:0];

DEST[MAXVL-1:256] := 0

**VPERMW (EVEX encoded versions)**

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

IF VL = 128 THEN n := 2; FI;

IF VL = 256 THEN n := 3; FI;

IF VL = 512 THEN n := 4; FI;

FOR j := 0 TO KL-1

i := j \* 16

id := 16\*SRC1[i+n:i]

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

THEN DEST[i+15:i] := SRC2[id+15:id]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

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

```

VPERMD __m512i __mm512_permutexvar_epi32( __m512i idx, __m512i a);
VPERMD __m512i __mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i __mm512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i __mm256_permutexvar_epi32( __m256i idx, __m256i a);
VPERMD __m256i __mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i __mm256_maskz_permutexvar_epi32(__mmask8 k, __m256i idx, __m256i a);
VPERMW __m512i __mm512_permutexvar_epi16( __m512i idx, __m512i a);
VPERMW __m512i __mm512_mask_permutexvar_epi16(__m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i __mm512_maskz_permutexvar_epi16(__mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i __mm256_permutexvar_epi16( __m256i idx, __m256i a);
VPERMW __m256i __mm256_mask_permutexvar_epi16(__m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i __mm256_maskz_permutexvar_epi16(__mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i __mm_permutexvar_epi16( __m128i idx, __m128i a);
VPERMW __m128i __mm_mask_permutexvar_epi16(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i __mm_maskz_permutexvar_epi16(__mmask8 k, __m128i idx, __m128i a);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

Non-EVEX-encoded instruction, see 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. |

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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.0F38.W0 75 /r<br>VPERMI2B xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | A         | V/V                          | AVX512VL<br>AVX512_VBMI | Permute bytes in xmm3/m128 and xmm2 using<br>byte indexes in xmm1 and store the byte results<br>in xmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 75 /r<br>VPERMI2B ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | A         | V/V                          | AVX512VL<br>AVX512_VBMI | Permute bytes in ymm3/m256 and ymm2 using<br>byte indexes in ymm1 and store the byte results<br>in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 75 /r<br>VPERMI2B zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | A         | V/V                          | AVX512_VBMI             | Permute bytes in zmm3/m512 and zmm2 using<br>byte indexes in zmm1 and store the byte results<br>in zmm1 using writemask k1. |

### Instruction Operand Encoding

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

### Description

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

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

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

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

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

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

IF VL = 128:

id := 3;

ELSE IF VL = 256:

id := 4;

ELSE IF VL = 512:

id := 5;

FI;

TMP\_DEST[VL-1:0] := DEST[VL-1:0];

FOR j := 0 TO KL-1

off := 8\*SRC1[j\*8 + id:j\*8];

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

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

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7:j\*8] := 0;

\*ELSE

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

FI;

ENDFOR

DEST[MAX\_VL-1:VL] := 0;

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

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

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

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

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

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

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

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

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

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

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."

## VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 75 /r<br>VPERMI2W xmm1 {k1}{z}, xmm2,<br>xmm3/m128          | A          | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.                                  |
| EVEX.256.66.0F38.W1 75 /r<br>VPERMI2W ymm1 {k1}{z}, ymm2,<br>ymm3/m256          | A          | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.                                  |
| EVEX.512.66.0F38.W1 75 /r<br>VPERMI2W zmm1 {k1}{z}, zmm2,<br>zmm3/m512          | A          | V/V                          | AVX512BW                 | Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.                                  |
| EVEX.128.66.0F38.W0 76 /r<br>VPERMI2D xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst  | B          | V/V                          | AVX512VL<br>AVX512F      | 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.                           |
| EVEX.256.66.0F38.W0 76 /r<br>VPERMI2D ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst  | B          | V/V                          | AVX512VL<br>AVX512F      | 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.                           |
| EVEX.512.66.0F38.W0 76 /r<br>VPERMI2D zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst  | B          | V/V                          | AVX512F                  | Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.                           |
| EVEX.128.66.0F38.W1 76 /r<br>VPERMI2Q xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst  | B          | V/V                          | AVX512VL<br>AVX512F      | 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.                             |
| EVEX.256.66.0F38.W1 76 /r<br>VPERMI2Q ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst  | B          | V/V                          | AVX512VL<br>AVX512F      | 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.                             |
| EVEX.512.66.0F38.W1 76 /r<br>VPERMI2Q zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst  | B          | V/V                          | AVX512F                  | Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.                             |
| EVEX.128.66.0F38.W0 77 /r<br>VPERMI2PS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 77 /r<br>VPERMI2PS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 77 /r<br>VPERMI2PS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | B          | V/V                          | AVX512F                  | Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1. |



| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 77 /r<br>VPERMI2PD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 77 /r<br>VPERMI2PD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W1 77 /r<br>VPERMI2PD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | B         | V/V                          | AVX512F                  | Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1. |

### Instruction Operand Encoding

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

#### Description

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

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

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

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table\_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same table can be reused for example for a second iteration, while the index elements are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****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:i]

IF k1[j] OR \*no writemask\*

THEN

DEST[i+15:i] = TMP\_DEST[i+id+1] ? SRC2[off+15:off]

: SRC1[off+15:off]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+15:i] := 0

FI

FI;

ENDFOR

DEST[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:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+31:i] := TMP\_DEST[i+id+1] ? SRC2[31:0]

: SRC1[off+31:off]

ELSE

DEST[i+31:i] := TMP\_DEST[i+id+1] ? SRC2[off+31:off]

: SRC1[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

```

**VPERMI2Q/VPERMI2PD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF VL = 128

id := 0

FI;

IF VL = 256

id := 1

FI;

IF VL = 512

id := 2

FI;

TMP\_DEST := DEST

FOR j := 0 TO KL-1

i := j \* 64

off := 64 \* TMP\_DEST[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN

DEST[i+63:i] := TMP\_DEST[i+id+1] ? SRC2[63:0]

: SRC1[off+63:off]

ELSE

DEST[i+63:i] := TMP\_DEST[i+id+1] ? SRC2[off+63:off]

: SRC1[off+63:off]

FI

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI

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);  
 VPERMI \_\_m256i \_mm256\_permutex2var\_epi32(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2D \_\_m256i \_mm256\_mask\_permutex2var\_epi32(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);  
 VPERMI2D \_\_m256i \_mm256\_mask2\_permutex2var\_epi32(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMI2D \_\_m256i \_mm256\_maskz\_permutex2var\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2D \_\_m128i \_mm\_permutex2var\_epi32(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2D \_\_m128i \_mm\_mask\_permutex2var\_epi32(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMI2D \_\_m128i \_mm\_mask2\_permutex2var\_epi32(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMI2D \_\_m128i \_mm\_maskz\_permutex2var\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2PD \_\_m512d \_mm512\_permutex2var\_pd(\_\_m512d a, \_\_m512i idx, \_\_m512d b);  
 VPERMI2PD \_\_m512d \_mm512\_mask\_permutex2var\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512i idx, \_\_m512d b);  
 VPERMI2PD \_\_m512d \_mm512\_mask2\_permutex2var\_pd(\_\_m512d a, \_\_m512i idx, \_\_mmask8 k, \_\_m512d b);  
 VPERMI2PD \_\_m512d \_mm512\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512i idx, \_\_m512d b);  
 VPERMI2PD \_\_m256d \_mm256\_permutex2var\_pd(\_\_m256d a, \_\_m256i idx, \_\_m256d b);  
 VPERMI2PD \_\_m256d \_mm256\_mask\_permutex2var\_pd(\_\_m256d a, \_\_mmask8 k, \_\_m256i idx, \_\_m256d b);  
 VPERMI2PD \_\_m256d \_mm256\_mask2\_permutex2var\_pd(\_\_m256d a, \_\_m256i idx, \_\_mmask8 k, \_\_m256d b);  
 VPERMI2PD \_\_m256d \_mm256\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m256d a, \_\_m256i idx, \_\_m256d b);  
 VPERMI2PD \_\_m128d \_mm\_permutex2var\_pd(\_\_m128d a, \_\_m128i idx, \_\_m128d b);  
 VPERMI2PD \_\_m128d \_mm\_mask\_permutex2var\_pd(\_\_m128d a, \_\_mmask8 k, \_\_m128i idx, \_\_m128d b);  
 VPERMI2PD \_\_m128d \_mm\_mask2\_permutex2var\_pd(\_\_m128d a, \_\_m128i idx, \_\_mmask8 k, \_\_m128d b);  
 VPERMI2PD \_\_m128d \_mm\_maskz\_permutex2var\_pd(\_\_mmask8 k, \_\_m128d a, \_\_m128i idx, \_\_m128d b);  
 VPERMI2PS \_\_m512 \_mm512\_permutex2var\_ps(\_\_m512 a, \_\_m512i idx, \_\_m512 b);  
 VPERMI2PS \_\_m512 \_mm512\_mask\_permutex2var\_ps(\_\_m512 a, \_\_mmask16 k, \_\_m512i idx, \_\_m512 b);  
 VPERMI2PS \_\_m512 \_mm512\_mask2\_permutex2var\_ps(\_\_m512 a, \_\_m512i idx, \_\_mmask16 k, \_\_m512 b);  
 VPERMI2PS \_\_m512 \_mm512\_maskz\_permutex2var\_ps(\_\_mmask16 k, \_\_m512 a, \_\_m512i idx, \_\_m512 b);  
 VPERMI2PS \_\_m256 \_mm256\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMI2PS \_\_m256 \_mm256\_mask\_permutex2var\_ps(\_\_m256 a, \_\_mmask8 k, \_\_m256i idx, \_\_m256 b);  
 VPERMI2PS \_\_m256 \_mm256\_mask2\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_mmask8 k, \_\_m256 b);  
 VPERMI2PS \_\_m256 \_mm256\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMI2PS \_\_m128 \_mm\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMI2PS \_\_m128 \_mm\_mask\_permutex2var\_ps(\_\_m128 a, \_\_mmask8 k, \_\_m128i idx, \_\_m128 b);  
 VPERMI2PS \_\_m128 \_mm\_mask2\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_mmask8 k, \_\_m128 b);  
 VPERMI2PS \_\_m128 \_mm\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMI2Q \_\_m512i \_mm512\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2Q \_\_m512i \_mm512\_mask\_permutex2var\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i idx, \_\_m512i b);  
 VPERMI2Q \_\_m512i \_mm512\_mask2\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_mmask8 k, \_\_m512i b);  
 VPERMI2Q \_\_m512i \_mm512\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMI2Q \_\_m256i \_mm256\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2Q \_\_m256i \_mm256\_mask\_permutex2var\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);  
 VPERMI2Q \_\_m256i \_mm256\_mask2\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMI2Q \_\_m256i \_mm256\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMI2Q \_\_m128i \_mm\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMI2Q \_\_m128i \_mm\_mask\_permutex2var\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMI2Q \_\_m128i \_mm\_mask2\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMI2Q \_\_m128i \_mm\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i 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—Permute In-Lane of Pairs of Double Precision Floating-Point Values

| Opcode/<br>Instruction   | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|---------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 0D /r<br>VPERMILPD xmm1, xmm2, xmm3/m128                        | A       | V/V                          | AVX                      | Permute double precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.                                |
| VEX.256.66.0F38.W0 0D /r<br>VPERMILPD ymm1, ymm2, ymm3/m256                        | A       | V/V                          | AVX                      | Permute double precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.                                |
| EVEX.128.66.0F38.W1 0D /r<br>VPERMILPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst    | C       | V/V                          | AVX512VL<br>AVX512F      | Permute double precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.  |
| EVEX.256.66.0F38.W1 0D /r<br>VPERMILPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst    | C       | V/V                          | AVX512VL<br>AVX512F      | Permute double precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.  |
| EVEX.512.66.0F38.W1 0D /r<br>VPERMILPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst    | C       | V/V                          | AVX512F                  | Permute double precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.  |
| VEX.128.66.0F3A.W0 05 /r ib<br>VPERMILPD xmm1, xmm2/m128, imm8                     | B       | V/V                          | AVX                      | Permute double precision floating-point values in xmm2/m128 using controls from imm8.   |
| VEX.256.66.0F3A.W0 05 /r ib<br>VPERMILPD ymm1, ymm2/m256, imm8                     | B       | V/V                          | AVX                      | Permute double precision floating-point values in ymm2/m256 using controls from imm8.   |
| EVEX.128.66.0F3A.W1 05 /r ib<br>VPERMILPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8 | D       | V/V                          | AVX512VL<br>AVX512F      | Permute double precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F3A.W1 05 /r ib<br>VPERMILPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8 | D       | V/V                          | AVX512VL<br>AVX512F      | Permute double precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F3A.W1 05 /r ib<br>VPERMILPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8 | D       | V/V                          | AVX512F                  | Permute double precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | N/A        | ModRM:reg (w) | ModRM:r/m (r) | N/A           | N/A       |
| C     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| D     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A           | N/A       |

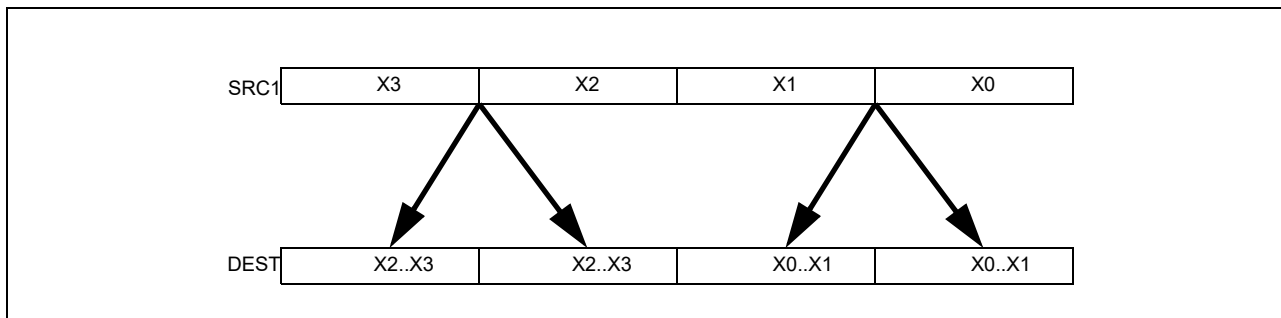
**Description**

(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).

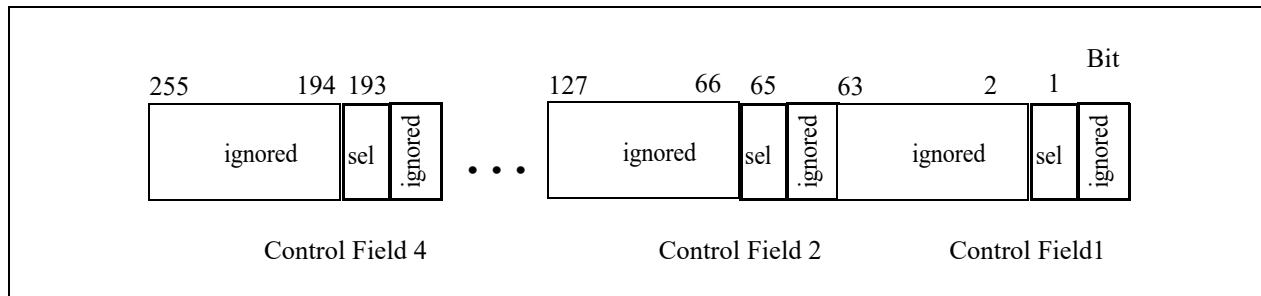
The control bits are located at bit 0 of each quadword element (see Figure 5-24). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

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**

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.



**Figure 5-24. VPERMILPD Shuffle Control**

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.





**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]
IF (imm8[1] = 1) THEN DEST[127:64] := SRC1[127:64]
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[1] = 1) THEN TMP_DEST[63:0] := SRC1[127:64]; FI;
IF (TMP_SRC2[65] = 0) THEN TMP_DEST[127:64] := SRC1[63:0]; FI;
IF (TMP_SRC2[65] = 1) THEN TMP_DEST[127:64] := SRC1[127:64]; FI;
IF VL >= 256
  IF (TMP_SRC2[129] = 0) THEN TMP_DEST[191:128] := SRC1[191:128]; FI;
  IF (TMP_SRC2[129] = 1) THEN TMP_DEST[191:128] := SRC1[255:192]; FI;
  IF (TMP_SRC2[193] = 0) THEN TMP_DEST[255:192] := SRC1[191:128]; FI;
  IF (TMP_SRC2[193] = 1) THEN TMP_DEST[255:192] := SRC1[255:192]; FI;
FI;
IF VL >= 512
  IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] := SRC1[319:256]; FI;
  IF (TMP_SRC2[257] = 1) THEN TMP_DEST[319:256] := SRC1[383:320]; FI;
  IF (TMP_SRC2[321] = 0) THEN TMP_DEST[383:320] := SRC1[319:256]; FI;
  IF (TMP_SRC2[321] = 1) THEN TMP_DEST[383:320] := SRC1[383:320]; FI;
  IF (TMP_SRC2[385] = 0) THEN TMP_DEST[447:384] := SRC1[447:384]; FI;
  IF (TMP_SRC2[385] = 1) THEN TMP_DEST[447:384] := SRC1[511:448]; FI;
  IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] := SRC1[447:384]; FI;
  IF (TMP_SRC2[449] = 1) THEN TMP_DEST[511:448] := SRC1[511:448]; FI;
FI;

```

```

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] := 0
        FI
      FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

**VPERMILPD (256-bit variable version)**

```

IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128] := SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128] := SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192] := SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192] := SRC1[255:192]
DEST[MAXVL-1:256] := 0

```

**VPERMILPD (128-bit variable version)**

```

IF (SRC2[1] = 0) THEN DEST[63:0] := SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0] := SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64] := SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64] := SRC1[127:64]
DEST[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_mask_permute_pd( __m256d s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d __mm256_maskz_permute_pd( __mmask8 k, __m256d a, int imm);
VPERMILPD __m128d __mm_mask_permute_pd( __m128d s, __mmask8 k, __m128d a, int imm);
VPERMILPD __m128d __mm_maskz_permute_pd( __mmask8 k, __m128d a, int imm);
VPERMILPD __m512d __mm512_permutevar_pd( __m512i i, __m512d a);
VPERMILPD __m512d __mm512_mask_permutevar_pd( __m512d s, __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m512d __mm512_maskz_permutevar_pd( __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m256d __mm256_mask_permutevar_pd( __m256d s, __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m256d __mm256_maskz_permutevar_pd( __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m128d __mm_mask_permutevar_pd( __m128d s, __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d __mm_maskz_permutevar_pd( __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d __mm_permute_pd( __m128d a, int control)
VPERMILPD __m256d __mm256_permute_pd( __m256d a, int control)
VPERMILPD __m128d __mm_permutevar_pd( __m128d a, __m128i control);
VPERMILPD __m256d __mm256_permutevar_pd( __m256d a, __m256i control);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see 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

| Opcode/<br>Instruction   | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|---------|------------------------------|--------------------------|--|
| VEX.128.66.0F38.W0 0C /r<br>VPERMILPS xmm1, xmm2, xmm3/m128                        | A       | V/V                          | AVX                      | Permute single-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.                             |
| VEX.128.66.0F3A.W0 04 /r ib<br>VPERMILPS xmm1, xmm2/m128, imm8                     | B       | V/V                          | AVX                      | Permute single-precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.                             |
| VEX.256.66.0F38.W0 0C /r<br>VPERMILPS ymm1, ymm2, ymm3/m256                        | A       | V/V                          | AVX                      | Permute single-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.                             |
| VEX.256.66.0F3A.W0 04 /r ib<br>VPERMILPS ymm1, ymm2/m256, imm8                     | B       | V/V                          | AVX                      | Permute single-precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.                             |
| EVEX.128.66.0F38.W0 0C /r<br>VPERMILPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst    | C       | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.  |
| EVEX.256.66.0F38.W0 0C /r<br>VPERMILPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst    | C       | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.  |
| EVEX.512.66.0F38.W0 0C /r<br>VPERMILPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst    | C       | V/V                          | AVX512F                  | Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.  |
| EVEX.128.66.0F3A.W0 04 /r ib<br>VPERMILPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8 | D       | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F3A.W0 04 /r ib<br>VPERMILPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8 | D       | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F3A.W0 04 /r<br>ibVPERMILPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8  | D       | V/V                          | AVX512F                  | Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | N/A        | ModRM:reg (w) | ModRM:r/m (r) | N/A           | N/A       |
| C     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| D     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A           | N/A       |

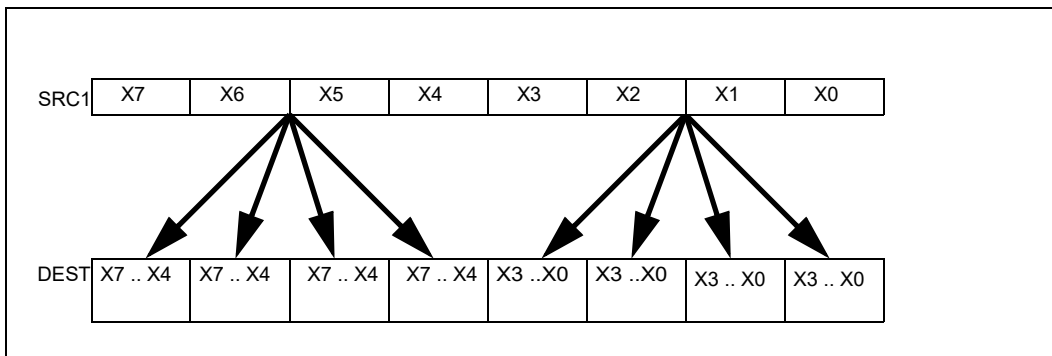
**Description**

Variable control version:

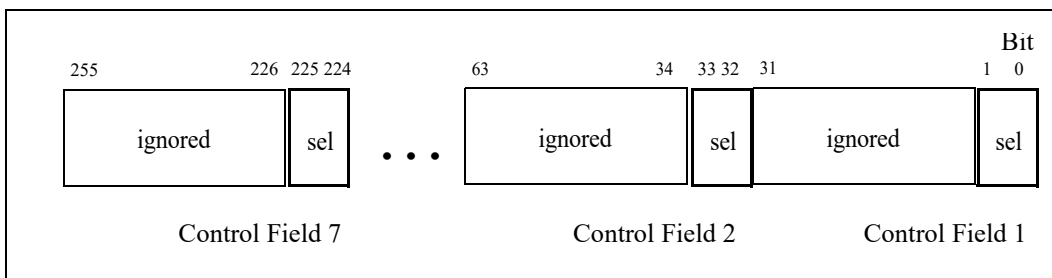
Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

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.



**Figure 5-25. VPERMILPS Operation**



**Figure 5-26. VPERMILPS Shuffle Control**

(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[63:32] := Select4(TMP_SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] := Select4(TMP_SRC1[127:0], imm8[5:4]);
TMP_DEST[127:96] := Select4(TMP_SRC1[127:0], imm8[7:6]); FI;
IF VL >= 256
  TMP_DEST[159:128] := Select4(TMP_SRC1[255:128], imm8[1:0]); FI;
  TMP_DEST[191:160] := Select4(TMP_SRC1[255:128], imm8[3:2]); FI;
  TMP_DEST[223:192] := Select4(TMP_SRC1[255:128], imm8[5:4]); FI;
  TMP_DEST[255:224] := Select4(TMP_SRC1[255:128], imm8[7:6]); FI;
FI;
IF VL >= 512
  TMP_DEST[287:256] := Select4(TMP_SRC1[383:256], imm8[1:0]); FI;
  TMP_DEST[319:288] := Select4(TMP_SRC1[383:256], imm8[3:2]); FI;
  TMP_DEST[351:320] := Select4(TMP_SRC1[383:256], imm8[5:4]); FI;
  TMP_DEST[383:352] := Select4(TMP_SRC1[383:256], imm8[7:6]); FI;
  TMP_DEST[415:384] := Select4(TMP_SRC1[511:384], imm8[1:0]); FI;
  TMP_DEST[447:416] := Select4(TMP_SRC1[511:384], imm8[3:2]); FI;
  TMP_DEST[479:448] := Select4(TMP_SRC1[511:384], imm8[5:4]); FI;
  TMP_DEST[511:480] := Select4(TMP_SRC1[511:384], imm8[7:6]); FI;
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[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] := Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] := Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] := Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] := Select4(SRC1[255:128], imm8[7:6]);

```

**VPERMILPS (128-bit immediate version)**

```

DEST[31:0] := Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] := Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] := Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] := Select4(SRC1[127:0], imm8[7:6]);
DEST[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

  TMP\_DEST[159:128] := Select4(SRC1[255:128], TMP\_SRC2[129:128]);

  TMP\_DEST[191:160] := Select4(SRC1[255:128], TMP\_SRC2[161:160]);

  TMP\_DEST[223:192] := Select4(SRC1[255:128], TMP\_SRC2[193:192]);

  TMP\_DEST[255:224] := Select4(SRC1[255:128], TMP\_SRC2[225:224]);

FI;

IF VL >= 512

  TMP\_DEST[287:256] := Select4(SRC1[383:256], TMP\_SRC2[257:256]);

  TMP\_DEST[319:288] := Select4(SRC1[383:256], TMP\_SRC2[289:288]);

  TMP\_DEST[351:320] := Select4(SRC1[383:256], TMP\_SRC2[321:320]);

  TMP\_DEST[383:352] := Select4(SRC1[383:256], TMP\_SRC2[353:352]);

  TMP\_DEST[415:384] := Select4(SRC1[511:384], TMP\_SRC2[385:384]);

  TMP\_DEST[447:416] := Select4(SRC1[511:384], TMP\_SRC2[417:416]);

  TMP\_DEST[479:448] := Select4(SRC1[511:384], TMP\_SRC2[449:448]);

  TMP\_DEST[511:480] := Select4(SRC1[511:384], TMP\_SRC2[481:480]);

FI;

FOR j := 0 TO KL-1

  i := j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] := TMP\_DEST[i+31:i]

    ELSE

      IF \*merging-masking\*

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE DEST[i+31:i] := 0 ;zeroing-masking

```

        FI;
    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[159:128] := Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] := Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] := Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] := Select4(SRC1[255:128], SRC2[225:224]);
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 __mm512_mask_permute_ps( __m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 __mm512_maskz_permute_ps( __mmask16 k, __m512 a, int imm);
VPERMILPS __m256 __mm256_mask_permute_ps( __m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 __mm256_maskz_permute_ps( __mmask8 k, __m256 a, int imm);
VPERMILPS __m128 __mm_mask_permute_ps( __m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 __mm_maskz_permute_ps( __mmask8 k, __m128 a, int imm);
VPERMILPS __m512 __mm512_permutevar_ps( __m512i i, __m512 a);
VPERMILPS __m512 __mm512_mask_permutevar_ps( __m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m512 __mm512_maskz_permutevar_ps( __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m256 __mm256_mask_permutevar_ps( __m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 __mm256_maskz_permutevar_ps( __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 __mm_mask_permutevar_ps( __m128 s, __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 __mm_maskz_permutevar_ps( __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 __mm_permute_ps( __m128 a, int control);
VPERMILPS __m256 __mm256_permute_ps( __m256 a, int control);
VPERMILPS __m128 __mm_permutevar_ps( __m128 a, __m128i control);
VPERMILPS __m256 __mm256_permutevar_ps( __m256 a, __m256i control);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD If VEX.W = 1.

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

Additionally:

#UD If either (E)VEX.vvvv != 1111B and with imm8.

## VPERMPD—Permute Double Precision Floating-Point Elements

| Opcode/<br>Instruction   | Op / En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|---------|-----------------------------|--------------------------|---|
| VEX.256.66.0F3A.W1 01 /r ib<br>VPERMPD ymm1, ymm2/m256, imm8                     | A       | V/V                         | AVX2                     | Permute double precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.                                 |
| EVEX.256.66.0F3A.W1 01 /r ib<br>VPERMPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8 | B       | V/V                         | AVX512VL<br>AVX512F      | Permute double precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1. |
| EVEX.512.66.0F3A.W1 01 /r ib<br>VPERMPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8 | B       | V/V                         | AVX512F                  | Permute double precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1. |
| EVEX.256.66.0F38.W1 16 /r<br>VPERMPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst    | C       | V/V                         | AVX512VL<br>AVX512F      | Permute double precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1. |
| EVEX.512.66.0F38.W1 16 /r<br>VPERMPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst    | C       | V/V                         | AVX512F                  | Permute double precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | ModRM:r/m (r) | imm8          | N/A       |
| B     | Full       | ModRM:reg (w) | ModRM:r/m (r) | imm8          | N/A       |
| C     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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 \* 64

  IF (EVEX.b = 1) AND (SRC \*is memory\*)  
    THEN TMP\_SRC[i+63:i] := SRC[63:0];  
    ELSE TMP\_SRC[i+63:i] := SRC[i+63:i];

FI;

ENDFOR;

TMP\_DEST[63:0] := (TMP\_SRC[256:0] &gt;&gt; (IMM8[1:0] \* 64))[63:0];

TMP\_DEST[127:64] := (TMP\_SRC[256:0] &gt;&gt; (IMM8[3:2] \* 64))[63:0];

TMP\_DEST[191:128] := (TMP\_SRC[256:0] &gt;&gt; (IMM8[5:4] \* 64))[63:0];

TMP\_DEST[255:192] := (TMP\_SRC[256:0] &gt;&gt; (IMM8[7:6] \* 64))[63:0];

IF VL &gt;= 512

TMP\_DEST[319:256] := (TMP\_SRC[511:256] &gt;&gt; (IMM8[1:0] \* 64))[63:0];

TMP\_DEST[383:320] := (TMP\_SRC[511:256] &gt;&gt; (IMM8[3:2] \* 64))[63:0];

TMP\_DEST[447:384] := (TMP\_SRC[511:256] &gt;&gt; (IMM8[5:4] \* 64))[63:0];

TMP\_DEST[511:448] := (TMP\_SRC[511:256] &gt;&gt; (IMM8[7:6] \* 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 (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j \* 64

  IF (EVEX.b = 1) AND (SRC2 \*is memory\*)  
    THEN TMP\_SRC2[i+63:i] := SRC2[63:0];  
    ELSE TMP\_SRC2[i+63:i] := SRC2[i+63:i];

FI;

ENDFOR;

IF VL = 256

TMP\_DEST[63:0] := (TMP\_SRC2[255:0] &gt;&gt; (SRC1[1:0] \* 64))[63:0];

TMP\_DEST[127:64] := (TMP\_SRC2[255:0] &gt;&gt; (SRC1[65:64] \* 64))[63:0];

TMP\_DEST[191:128] := (TMP\_SRC2[255:0] &gt;&gt; (SRC1[129:128] \* 64))[63:0];

TMP\_DEST[255:192] := (TMP\_SRC2[255:0] &gt;&gt; (SRC1[193:192] \* 64))[63:0];

FI;

IF VL = 512

TMP\_DEST[63:0] := (TMP\_SRC2[511:0] &gt;&gt; (SRC1[2:0] \* 64))[63:0];

```

TMP_DEST[127:64] := (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
TMP_DEST[191:128] := (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
TMP_DEST[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
TMP_DEST[319:256] := (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
TMP_DEST[383:320] := (TMP_SRC2[511:0] >> (SRC1[322:320] * 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[191:128] := (SRC[255:0] >> (IMM8[5:4] * 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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| VEX.256.66.0F38.W0 16 /r<br>VPERMPS ymm1, ymm2,<br>ymm3/m256                  | A          | V/V                          | AVX2                     | Permute single-precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.                                  |
| EVEX.256.66.0F38.W0 16 /r<br>VPERMPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | B          | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F38.W0 16 /r<br>VPERMPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | B          | V/V                          | AVX512F                  | Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.   |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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];

  TMP\_DEST[159:128] := (TMP\_SRC2[255:0] >> (SRC1[130:128] \* 32))[31:0];

  TMP\_DEST[191:160] := (TMP\_SRC2[255:0] >> (SRC1[162:160] \* 32))[31:0];

  TMP\_DEST[223:192] := (TMP\_SRC2[255:0] >> (SRC1[193:192] \* 32))[31:0];

```

    TMP_DEST[255:224] := (TMP_SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
FI;
IF VL = 512
    TMP_DEST[31:0] := (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
    TMP_DEST[63:32] := (TMP_SRC2[511:0] >> (SRC1[35:32] * 32))[31:0];
    TMP_DEST[95:64] := (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
    TMP_DEST[127:96] := (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
    TMP_DEST[159:128] := (TMP_SRC2[511:0] >> (SRC1[131:128] * 32))[31:0];
    TMP_DEST[191:160] := (TMP_SRC2[511:0] >> (SRC1[163:160] * 32))[31:0];
    TMP_DEST[223:192] := (TMP_SRC2[511:0] >> (SRC1[195:192] * 32))[31:0];
    TMP_DEST[255:224] := (TMP_SRC2[511:0] >> (SRC1[227:224] * 32))[31:0];
    TMP_DEST[287:256] := (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
    TMP_DEST[319:288] := (TMP_SRC2[511:0] >> (SRC1[291:288] * 32))[31:0];
    TMP_DEST[351:320] := (TMP_SRC2[511:0] >> (SRC1[323:320] * 32))[31:0];
    TMP_DEST[383:352] := (TMP_SRC2[511:0] >> (SRC1[355:352] * 32))[31:0];
    TMP_DEST[415:384] := (TMP_SRC2[511:0] >> (SRC1[387:384] * 32))[31:0];
    TMP_DEST[447:416] := (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
    TMP_DEST[479:448] := (TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
    TMP_DEST[511:480] := (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];
FI;
FOR j := 0 TO KL-1
    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 ;zeroing-masking
    FI;
FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

**VPERMPS (VEX.256 encoded version)**

```

DEST[31:0] := (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] := (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] := (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] := (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] := (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] := (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] := (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] := (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] := 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPERMPS __m512 __mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 __mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 __mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 __mm256_permutexvar_ps(__m256 i, __m256 a);
VPERMPS __m256 __mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMPS __m256 __mm256_maskz_permutexvar_ps(__mmask8 k, __m256 i, __m256 a);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

Non-EVEX-encoded instruction, see Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

#UD                      If VEX.L = 0.

EVEX-encoded instruction, see Table 2-50, “Type E4NF Class Exception Conditions.”

## VPERMQ—Qwords Element Permutation

| Opcode/<br>Instruction  | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|------------------------------|--------------------------|---|
| VEX.256.66.0F3A.W1 00 /r ib<br>VPERMQ ymm1, ymm2/m256, imm8                     | A       | V/V                          | AVX2                     | Permute qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.         |
| EVEX.256.66.0F3A.W1 00 /r ib<br>VPERMQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8 | B       | V/V                          | AVX512VL<br>AVX512F      | Permute qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1. |
| EVEX.512.66.0F3A.W1 00 /r ib<br>VPERMQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8 | B       | V/V                          | AVX512F                  | Permute qwords in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1. |
| EVEX.256.66.0F38.W1 36 /r<br>VPERMQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst    | C       | V/V                          | AVX512VL<br>AVX512F      | Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1. |
| EVEX.512.66.0F38.W1 36 /r<br>VPERMQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst    | C       | V/V                          | AVX512F                  | Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | ModRM:r/m (r) | imm8          | N/A       |
| B     | Full       | ModRM:reg (w) | ModRM:r/m (r) | imm8          | N/A       |
| C     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.

**Operation****VPERMQ (EVEX - imm8 control forms)**

(KL, VL) = (4, 256), (8, 512)

```

FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC *is memory*)
    THEN TMP_SRC[i+63:i] := SRC[63:0];
    ELSE TMP_SRC[i+63:i] := SRC[i+63:i];
  FI;
ENDFOR;
TMP_DEST[63:0] := (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] := (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[191:128] := (TMP_SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
TMP_DEST[255:192] := (TMP_SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
  TMP_DEST[319:256] := (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
  TMP_DEST[383:320] := (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
  TMP_DEST[447:384] := (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
  TMP_DEST[511:448] := (TMP_SRC[511:256] >> (IMM8[7:6] * 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;
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

**VPERMQ (EVEX - vector control forms)**

(KL, VL) = (4, 256), (8, 512)

```

FOR j := 0 TO KL-1
  i := j * 64
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+63:i] := SRC2[63:0];
    ELSE TMP_SRC2[i+63:i] := SRC2[i+63:i];
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[63:0] := (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
  TMP_DEST[127:64] := (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
  TMP_DEST[191:128] := (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
  TMP_DEST[255:192] := (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
  TMP_DEST[63:0] := (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
  TMP_DEST[127:64] := (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
  TMP_DEST[191:128] := (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
  TMP_DEST[255:192] := (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];

```

```

TMP_DEST[319:256] := (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
TMP_DEST[383:320] := (TMP_SRC2[511:0] >> (SRC1[322:320] * 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[191:128] := (SRC[255:0] >> (IMM8[5:4] * 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(__m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i __mm512_maskz_permutex_epi64(__mmask8 k, __m512i a, int imm);
VPERMQ __m512i __mm512_permutexvar_epi64(__m512i a, __m512i b);
VPERMQ __m512i __mm512_mask_permutexvar_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i __mm512_maskz_permutexvar_epi64(__mmask8 k, __m512i a, __m512i b);
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 EVEX.vvvv != 1111B and with imm8.

```



## VPERMT2B—Full Permute of Bytes From Two Tables Overwriting a Table

| Opcode/<br>Instruction   | Op<br>/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|--|---------------|------------------------------|-------------------------|---|
| EVEX.128.66.0F38.W0 7D /r<br>VPERMT2B xmm1 {k1}{z}, xmm2,<br>xmm3/m128 | A             | V/V                          | AVX512VL<br>AVX512_VBMI | Permute bytes in xmm3/m128 and xmm1 using byte indexes in xmm2 and store the byte results in xmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 7D /r<br>VPERMT2B ymm1 {k1}{z}, ymm2,<br>ymm3/m256 | A             | V/V                          | AVX512VL<br>AVX512_VBMI | Permute bytes in ymm3/m256 and ymm1 using byte indexes in ymm2 and store the byte results in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 7D /r<br>VPERMT2B zmm1 {k1}{z}, zmm2,<br>zmm3/m512 | A             | V/V                          | AVX512_VBMI             | Permute bytes in zmm3/m512 and zmm1 using byte indexes in zmm2 and store the byte results in zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | Full Mem   | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Permutes byte values from two tables, comprising of the first operand (also the destination operand) and the third operand (the second source operand). The second operand (the first source operand) provides byte indices to select byte results from the two tables. The selected byte elements are written to the destination at byte granularity under the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result. The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. In each index byte, the id bit for table selection is bit 6/5/4, and bits [5:0]/[4:0]/[3:0] selects element within each input table.

Note that these instructions permit a byte value in the source operands to be copied to more than one location in the destination operand. Also, the second table and the indices can be reused in subsequent iterations, but the first table is overwritten.

Bits (MAX\_VL-1:256/128) of the destination are zeroed for VL=256,128.

**Operation****VPERMT2B (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

IF VL = 128:

id := 3;

ELSE IF VL = 256:

id := 4;

ELSE IF VL = 512:

id := 5;

FI;

TMP\_DEST[VL-1:0] := DEST[VL-1:0];

FOR j := 0 TO KL-1

off := 8\*SRC1[j\*8 + id:j\*8];

IF k1[j] OR \*no writemask\*:

DEST[j\*8 + 7:j\*8] := SRC1[j\*8+id+1]? SRC2[off+7:off] : TMP\_DEST[off+7:off];

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7:j\*8] := 0;

\*ELSE

DEST[j\*8 + 7:j\*8] remains unchanged\*

FI;

ENDFOR

DEST[MAX\_VL-1:VL] := 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPERMT2B \_\_m512i \_\_mm512\_permutex2var\_epi8(\_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m512i \_\_mm512\_mask\_permutex2var\_epi8(\_\_m512i a, \_\_mmask64 k, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi8(\_\_mmask64 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);

VPERMT2B \_\_m256i \_\_mm256\_permutex2var\_epi8(\_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m256i \_\_mm256\_mask\_permutex2var\_epi8(\_\_m256i a, \_\_mmask32 k, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi8(\_\_mmask32 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);

VPERMT2B \_\_m128i \_\_mm\_permutex2var\_epi8(\_\_m128i a, \_\_m128i idx, \_\_m128i b);

VPERMT2B \_\_m128i \_\_mm\_mask\_permutex2var\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m128i idx, \_\_m128i b);

VPERMT2B \_\_m128i \_\_mm\_maskz\_permutex2var\_epi8(\_\_mmask16 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb in Table 2-50, "Type E4NF Class Exception Conditions."

## VPERMT2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting One Table

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 7D /r<br>VPERMT2W xmm1 {k1}{z}, xmm2,<br>xmm3/m128          | A         | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.                                  |
| EVEX.256.66.0F38.W1 7D /r<br>VPERMT2W ymm1 {k1}{z}, ymm2,<br>ymm3/m256          | A         | V/V                          | AVX512VL<br>AVX512BW     | Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.                                  |
| EVEX.512.66.0F38.W1 7D /r<br>VPERMT2W zmm1 {k1}{z}, zmm2,<br>zmm3/m512          | A         | V/V                          | AVX512BW                 | Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.                                  |
| EVEX.128.66.0F38.W0 7E /r<br>VPERMT2D xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst  | B         | V/V                          | AVX512VL<br>AVX512F      | Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.                           |
| EVEX.256.66.0F38.W0 7E /r<br>VPERMT2D ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst  | B         | V/V                          | AVX512VL<br>AVX512F      | Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.                           |
| EVEX.512.66.0F38.W0 7E /r<br>VPERMT2D zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst  | B         | V/V                          | AVX512F                  | Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.                           |
| EVEX.128.66.0F38.W1 7E /r<br>VPERMT2Q xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst  | B         | V/V                          | AVX512VL<br>AVX512F      | Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.                             |
| EVEX.256.66.0F38.W1 7E /r<br>VPERMT2Q ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst  | B         | V/V                          | AVX512VL<br>AVX512F      | Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.                             |
| EVEX.512.66.0F38.W1 7E /r<br>VPERMT2Q zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst  | B         | V/V                          | AVX512F                  | Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.                             |
| EVEX.128.66.0F38.W0 7F /r<br>VPERMT2PS xmm1 {k1}{z},<br>xmm2, xmm3/m128/m32bcst | B         | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 7F /r<br>VPERMT2PS ymm1 {k1}{z},<br>ymm2, ymm3/m256/m32bcst | B         | V/V                          | AVX512VL<br>AVX512F      | Permute single-precision floating-point values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 7F /r<br>VPERMT2PS zmm1 {k1}{z},<br>zmm2, zmm3/m512/m32bcst | B         | V/V                          | AVX512F                  | Permute single-precision floating-point values from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1. |
| EVEX.128.66.0F38.W1 7F /r<br>VPERMT2PD xmm1 {k1}{z},<br>xmm2, xmm3/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | Permute double precision floating-point values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 7F /r<br>VPERMT2PD ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | Permute double precision floating-point values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1. |
| EVEX.512.66.0F38.W1 7F /r<br>VPERMT2PD zmm1 {k1}{z},<br>zmm2, zmm3/m512/m64bcst | B         | V/V                          | AVX512F                  | Permute double precision floating-point values from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1. |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | Full Mem   | ModRM:reg (r,w)  | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## Description

Permutates 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table\_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table\_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

## Operation

**VPERMT2W (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

IF VL = 128

id := 2

FI;

IF VL = 256

id := 3

FI;

IF VL = 512

id := 4

FI;

TMP\_DEST := DEST

FOR j := 0 TO KL-1

i := j \* 16

off := 16\*SRC1[i+id:i]

IF k1[j] OR \*no writemask\*

THEN

DEST[i+15:i]=SRC1[i+id+1] ? SRC2[off+15:off]  
: TMP\_DEST[off+15:off]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+15:i] remains unchanged\*

ELSE ; zeroing-masking

```

                DEST[i+15:i] := 0
            FI
        FI;
    ENDFOR
    DEST[MAXVL-1:VL] := 0

VPERMT2D/VPERMT2PS (EVEX encoded versions)
    (KL, VL) = (4, 128), (8, 256), (16, 512)
    IF VL = 128
        id := 1
    FI;
    IF VL = 256
        id := 2
    FI;
    IF VL = 512
        id := 3
    FI;
    TMP_DEST := DEST
    FOR j := 0 TO KL-1
        i := j * 32
        off := 32*SRC1[i+id:i]
        IF k1[j] OR *no writemask*
            THEN
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN
                        DEST[i+31:i] := SRC1[i+id+1] ? SRC2[31:0]
                        : TMP_DEST[off+31:off]
                    ELSE
                        DEST[i+31:i] := SRC1[i+id+1] ? SRC2[off+31:off]
                        : TMP_DEST[off+31:off]
                    FI
                ELSE
                    IF *merging-masking* ; merging-masking
                        THEN *DEST[i+31:i] remains unchanged*
                    ELSE ; zeroing-masking
                        DEST[i+31:i] := 0
                    FI
                FI
            ELSE
                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[j+id:i]
IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        DEST[j+63:i] := SRC1[j+id+1] ? SRC2[63:0]
        : TMP_DEST[off+63:off]
      ELSE
        DEST[j+63:i] := SRC1[j+id+1] ? SRC2[off+63:off]
        : TMP_DEST[off+63:off]
      FI
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[j+63:i] := 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, __m512i idx, __m512d b);
VPERMT2PD __m512d __mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMT2PD __m512d __mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMT2PD __m512d __mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMT2PD __m256d __mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMT2PD __m256d __mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMT2PD __m256d __mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMT2PD __m256d __mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMT2PD __m128d __mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMT2PD __m128d __mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMT2PD __m128d __mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMT2PD __m128d __mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMT2PS __m512 __mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMT2PS __m512 __mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMT2PS __m512 __mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMT2PS __m512 __mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);

```

VPERMT2PS \_\_m256 \_\_mm256\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_mask\_permutex2var\_ps(\_\_m256 a, \_\_mmask8 k, \_\_m256i idx, \_\_m256 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_mask2\_permutex2var\_ps(\_\_m256 a, \_\_m256i idx, \_\_mmask8 k, \_\_m256 b);  
 VPERMT2PS \_\_m256 \_\_mm256\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m256 a, \_\_m256i idx, \_\_m256 b);  
 VPERMT2PS \_\_m128 \_\_mm\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMT2PS \_\_m128 \_\_mm\_mask\_permutex2var\_ps(\_\_m128 a, \_\_mmask8 k, \_\_m128i idx, \_\_m128 b);  
 VPERMT2PS \_\_m128 \_\_mm\_mask2\_permutex2var\_ps(\_\_m128 a, \_\_m128i idx, \_\_mmask8 k, \_\_m128 b);  
 VPERMT2PS \_\_m128 \_\_mm\_maskz\_permutex2var\_ps(\_\_mmask8 k, \_\_m128 a, \_\_m128i idx, \_\_m128 b);  
 VPERMT2Q \_\_m512i \_\_mm512\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2Q \_\_m512i \_\_mm512\_mask\_permutex2var\_epi64(\_\_m512i a, \_\_mmask8 k, \_\_m512i idx, \_\_m512i b);  
 VPERMT2Q \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi64(\_\_m512i a, \_\_m512i idx, \_\_mmask8 k, \_\_m512i b);  
 VPERMT2Q \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_mask\_permutex2var\_epi64(\_\_m256i a, \_\_mmask8 k, \_\_m256i idx, \_\_m256i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi64(\_\_m256i a, \_\_m256i idx, \_\_mmask8 k, \_\_m256i b);  
 VPERMT2Q \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2Q \_\_m128i \_\_mm\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2Q \_\_m128i \_\_mm\_mask\_permutex2var\_epi64(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMT2Q \_\_m128i \_\_mm\_mask2\_permutex2var\_epi64(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMT2Q \_\_m128i \_\_mm\_maskz\_permutex2var\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2W \_\_m512i \_\_mm512\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2W \_\_m512i \_\_mm512\_mask\_permutex2var\_epi16(\_\_m512i a, \_\_mmask32 k, \_\_m512i idx, \_\_m512i b);  
 VPERMT2W \_\_m512i \_\_mm512\_mask2\_permutex2var\_epi16(\_\_m512i a, \_\_m512i idx, \_\_mmask32 k, \_\_m512i b);  
 VPERMT2W \_\_m512i \_\_mm512\_maskz\_permutex2var\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i idx, \_\_m512i b);  
 VPERMT2W \_\_m256i \_\_mm256\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2W \_\_m256i \_\_mm256\_mask\_permutex2var\_epi16(\_\_m256i a, \_\_mmask16 k, \_\_m256i idx, \_\_m256i b);  
 VPERMT2W \_\_m256i \_\_mm256\_mask2\_permutex2var\_epi16(\_\_m256i a, \_\_m256i idx, \_\_mmask16 k, \_\_m256i b);  
 VPERMT2W \_\_m256i \_\_mm256\_maskz\_permutex2var\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i idx, \_\_m256i b);  
 VPERMT2W \_\_m128i \_\_mm\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_m128i b);  
 VPERMT2W \_\_m128i \_\_mm\_mask\_permutex2var\_epi16(\_\_m128i a, \_\_mmask8 k, \_\_m128i idx, \_\_m128i b);  
 VPERMT2W \_\_m128i \_\_mm\_mask2\_permutex2var\_epi16(\_\_m128i a, \_\_m128i idx, \_\_mmask8 k, \_\_m128i b);  
 VPERMT2W \_\_m128i \_\_mm\_maskz\_permutex2var\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i idx, \_\_m128i b);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

VPERMT2D/Q/PS/PD: See 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

| Opcode/<br>Instruction                                   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag    | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 62 /r<br>VPEXPANDB xmm1{k1}{z}, m128 | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Expands up to 128 bits of packed byte values from m128 to xmm1 with writemask k1.         |
| EVEX.128.66.0F38.W0 62 /r<br>VPEXPANDB xmm1{k1}{z}, xmm2 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Expands up to 128 bits of packed byte values from xmm2 to xmm1 with writemask k1.         |
| EVEX.256.66.0F38.W0 62 /r<br>VPEXPANDB ymm1{k1}{z}, m256 | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Expands up to 256 bits of packed byte values from m256 to ymm1 with writemask k1.         |
| EVEX.256.66.0F38.W0 62 /r<br>VPEXPANDB ymm1{k1}{z}, ymm2 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Expands up to 256 bits of packed byte values from ymm2 to ymm1 with writemask k1.         |
| EVEX.512.66.0F38.W0 62 /r<br>VPEXPANDB zmm1{k1}{z}, m512 | A         | V/V                          | AVX512_VBMI2             | Expands up to 512 bits of packed byte values from m512 to zmm1 with writemask k1.         |
| EVEX.512.66.0F38.W0 62 /r<br>VPEXPANDB zmm1{k1}{z}, zmm2 | B         | V/V                          | AVX512_VBMI2             | Expands up to 512 bits of packed byte values from zmm2 to zmm1 with writemask k1.         |
| EVEX.128.66.0F38.W1 62 /r<br>VPEXPANDW xmm1{k1}{z}, m128 | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Expands up to 128 bits of packed word values from m128 to xmm1 with writemask k1.         |
| EVEX.128.66.0F38.W1 62 /r<br>VPEXPANDW xmm1{k1}{z}, xmm2 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Expands up to 128 bits of packed word values from xmm2 to xmm1 with writemask k1.         |
| EVEX.256.66.0F38.W1 62 /r<br>VPEXPANDW ymm1{k1}{z}, m256 | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Expands up to 256 bits of packed word values from m256 to ymm1 with writemask k1.         |
| EVEX.256.66.0F38.W1 62 /r<br>VPEXPANDW ymm1{k1}{z}, ymm2 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Expands up to 256 bits of packed word values from ymm2 to ymm1 with writemask k1.         |
| EVEX.512.66.0F38.W1 62 /r<br>VPEXPANDW zmm1{k1}{z}, m512 | A         | V/V                          | AVX512_VBMI2             | Expands up to 512 bits of packed word values from m512 to zmm1 with writemask k1.         |
| EVEX.512.66.0F38.W1 62 /r<br>VPEXPANDW zmm1{k1}{z}, zmm2 | B         | V/V                          | AVX512_VBMI2             | Expands up to 512 bits of packed byte integer values from zmm2 to zmm1 with writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple         | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| B     | N/A           | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

#### Description

Expands (loads) up to 64 byte integer values or 32 word integer values from the source operand (memory operand) to the destination operand (register operand), based on the active elements determined by the writemask operand.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Moves 128, 256 or 512 bits of packed byte integer values from the source operand (memory operand) to the destination operand (register operand). This instruction is used to load from an int8 vector register or memory location while inserting the data into sparse elements of destination vector register using the active elements pointed out by the operand writemask.

This instruction supports memory fault suppression.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.



**Operation****VPEXPANDB**

(KL, VL) = (16, 128), (32, 256), (64, 512)

k := 0

FOR j := 0 TO KL-1:

IF k1[j] OR \*no writemask\*:

DEST.byte[j] := SRC.byte[k];

k := k + 1

ELSE:

IF \*merging-masking\*:

\*DEST.byte[j] remains unchanged\*

ELSE: ; zeroing-masking

DEST.byte[j] := 0

DEST[MAX\_VL-1:VL] := 0

**VPEXPANDW**

(KL, VL) = (8, 128), (16, 256), (32, 512)

k := 0

FOR j := 0 TO KL-1:

IF k1[j] OR \*no writemask\*:

DEST.word[j] := SRC.word[k];

k := k + 1

ELSE:

IF \*merging-masking\*:

\*DEST.word[j] remains unchanged\*

ELSE: ; zeroing-masking

DEST.word[j] := 0

DEST[MAX\_VL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPEXPAND __m128i_mm_mask_expand_epi8(__m128i, __mmask16, __m128i);
VPEXPAND __m128i_mm_maskz_expand_epi8(__mmask16, __m128i);
VPEXPAND __m128i_mm_mask_expandloadu_epi8(__m128i, __mmask16, const void*);
VPEXPAND __m128i_mm_maskz_expandloadu_epi8(__mmask16, const void*);
VPEXPAND __m256i_mm256_mask_expand_epi8(__m256i, __mmask32, __m256i);
VPEXPAND __m256i_mm256_maskz_expand_epi8(__mmask32, __m256i);
VPEXPAND __m256i_mm256_mask_expandloadu_epi8(__m256i, __mmask32, const void*);
VPEXPAND __m256i_mm256_maskz_expandloadu_epi8(__mmask32, const void*);
VPEXPAND __m512i_mm512_mask_expand_epi8(__m512i, __mmask64, __m512i);
VPEXPAND __m512i_mm512_maskz_expand_epi8(__mmask64, __m512i);
VPEXPAND __m512i_mm512_mask_expandloadu_epi8(__m512i, __mmask64, const void*);
VPEXPAND __m512i_mm512_maskz_expandloadu_epi8(__mmask64, const void*);
VPEXPANDW __m128i_mm_mask_expand_epi16(__m128i, __mmask8, __m128i);
VPEXPANDW __m128i_mm_maskz_expand_epi16(__mmask8, __m128i);
VPEXPANDW __m128i_mm_mask_expandloadu_epi16(__m128i, __mmask8, const void*);
VPEXPANDW __m128i_mm_maskz_expandloadu_epi16(__mmask8, const void*);
VPEXPANDW __m256i_mm256_mask_expand_epi16(__m256i, __mmask16, __m256i);
VPEXPANDW __m256i_mm256_maskz_expand_epi16(__mmask16, __m256i);
VPEXPANDW __m256i_mm256_mask_expandloadu_epi16(__m256i, __mmask16, const void*);
VPEXPANDW __m256i_mm256_maskz_expandloadu_epi16(__mmask16, const void*);
VPEXPANDW __m512i_mm512_mask_expand_epi16(__m512i, __mmask32, __m512i);
VPEXPANDW __m512i_mm512_maskz_expand_epi16(__mmask32, __m512i);
VPEXPANDW __m512i_mm512_mask_expandloadu_epi16(__m512i, __mmask32, const void*);
VPEXPANDW __m512i_mm512_maskz_expandloadu_epi16(__mmask32, const void*);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, “Type E4 Class Exception Conditions.”

## VPEXPANDD—Load Sparse Packed Doubleword Integer Values From Dense Memory/Register

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 89 /r<br>VPEXPANDD xmm1 {k1}{z},<br>xmm2/m128 | A         | V/V                          | AVX512VL<br>AVX512F      | Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.66.0F38.W0 89 /r<br>VPEXPANDD ymm1 {k1}{z},<br>ymm2/m256 | A         | V/V                          | AVX512VL<br>AVX512F      | Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.66.0F38.W0 89 /r<br>VPEXPANDD zmm1 {k1}{z},<br>zmm2/m512 | A         | V/V                          | AVX512F                  | Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

### Operation

#### VPEXPANDD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

k := 0

FOR j := 0 TO KL-1

  i := j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      DEST[i+31:i] := SRC[k+31:k];

      k := k + 32

    ELSE

      IF \*merging-masking\* ; merging-masking

        THEN \*DEST[i+31:i] remains unchanged\*

        ELSE ; zeroing-masking

          DEST[i+31:i] := 0

    FI

  FI;

ENDFOR

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VEXPANDD __m512i __mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VEXPANDD __m512i __mm512_maskz_expandloadu_epi32(__mmask16 k, void * a);
VEXPANDD __m512i __mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VEXPANDD __m512i __mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
VEXPANDD __m256i __mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VEXPANDD __m256i __mm256_maskz_expandloadu_epi32(__mmask8 k, void * a);
VEXPANDD __m256i __mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VEXPANDD __m256i __mm256_maskz_expand_epi32(__mmask8 k, __m256i a);
VEXPANDD __m128i __mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VEXPANDD __m128i __mm_maskz_expandloadu_epi32(__mmask8 k, void * a);
VEXPANDD __m128i __mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VEXPANDD __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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 89 /r<br>VPEXPANDQ xmm1 {k1}{z}, xmm2/m128 | A         | V/V                          | AVX512VL<br>AVX512F      | Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 89 /r<br>VPEXPANDQ ymm1 {k1}{z}, ymm2/m256 | A         | V/V                          | AVX512VL<br>AVX512F      | Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1. |
| EVEX.512.66.0F38.W1 89 /r<br>VPEXPANDQ zmm1 {k1}{z}, zmm2/m512 | A         | V/V                          | AVX512F                  | Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|---------------|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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: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**

```

VEXPANDQ __m512i __mm512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VEXPANDQ __m512i __mm512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VEXPANDQ __m512i __mm512_mask_expand_epi64(__m512i s, __mmask8 k, __m512i a);
VEXPANDQ __m512i __mm512_maskz_expand_epi64(__mmask8 k, __m512i a);
VEXPANDQ __m256i __mm256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VEXPANDQ __m256i __mm256_maskz_expandloadu_epi64(__mmask8 k, void * a);
VEXPANDQ __m256i __mm256_mask_expand_epi64(__m256i s, __mmask8 k, __m256i a);
VEXPANDQ __m256i __mm256_maskz_expand_epi64(__mmask8 k, __m256i a);
VEXPANDQ __m128i __mm_mask_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VEXPANDQ __m128i __mm_maskz_expandloadu_epi64(__mmask8 k, void * a);
VEXPANDQ __m128i __mm_mask_expand_epi64(__m128i s, __mmask8 k, __m128i a);
VEXPANDQ __m128i __mm_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.

## VPGATHERDD/VPGATHERQD—Gather Packed Dword Values Using Signed Dword/Qword Indices

| Opcode/<br>Instruction                                   | Op/<br>En | 64/32<br>-bit<br>Mode | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|-----------------------|--------------------------|--|
| VEX.128.66.0F38.W0 90 /r<br>VPGATHERDD xmm1, vm32x, xmm2 | RMV       | V/V                   | AVX2                     | Using dword indices specified in vm32x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |
| VEX.128.66.0F38.W0 91 /r<br>VPGATHERQD xmm1, vm64x, xmm2 | RMV       | V/V                   | AVX2                     | Using qword indices specified in vm64x, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |
| VEX.256.66.0F38.W0 90 /r<br>VPGATHERDD ymm1, vm32y, ymm2 | RMV       | V/V                   | AVX2                     | Using dword indices specified in vm32y, gather dword from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1.        |
| VEX.256.66.0F38.W0 91 /r<br>VPGATHERQD xmm1, vm64y, xmm2 | RMV       | V/V                   | AVX2                     | Using qword indices specified in vm64y, gather dword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |

### Instruction Operand Encoding

| Op/En | Operand 1       | Operand 2   | Operand 3       | Operand 4 |
|-------|-----------------|---|-----------------|-----------|
| RMV   | ModRM:reg (r,w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | VEX.vvvv (r, w) | N/A       |

### 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.

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[j + 31:i] := FFFFFFFFH; // extend from most significant bit

  ELSE

    MASK[j + 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[j + 31:i] := FETCH\_32BITS(DATA\_ADDR); // a fault exits the instruction

  FI;

  MASK[j + 31:i] := 0;

ENDFOR

DEST[MAXVL-1:128] := 0;



**VPGATHERQD (VEX.128 version)**

```

MASK[MAXVL-1:64] := 0;
FOR j := 0 to 3
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 1
  k := j * 64;
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:64] := 0;

```

**VPGATHERDD (VEX.256 version)**

```

MASK[MAXVL-1:256] := 0;
FOR j := 0 to 7
  i := j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 7
  i := j * 32;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] := 0;
ENDFOR
DEST[MAXVL-1:256] := 0;

```

**VPGATHERQD (VEX.256 version)**

```

MASK[MAXVL-1:128] := 0;
FOR j := 0 to 7
  i := j * 32;
  IF MASK[31:i] THEN
    MASK[j + 31:i] := FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[j + 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[j + 31:i] := FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[j + 31:i] := 0;
ENDFOR
DEST[MAXVL-1:128] := 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPGATHERDD: `__m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);`

VPGATHERDD: `__m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);`

VPGATHERDD: `__m256i _mm256_i32gather_epi32 ( int const * base, __m256i index, const int scale);`

VPGATHERDD: `__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: `__m128i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);`

VPGATHERQD: `__m128i _mm256_mask_i64gather_epi32 (__m128i 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

| Opcode/<br>Instruction                                      | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 90 /vsib<br>VPGATHERDD xmm1 {k1}, vm32x | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.256.66.0F38.W0 90 /vsib<br>VPGATHERDD ymm1 {k1}, vm32y | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.512.66.0F38.W0 90 /vsib<br>VPGATHERDD zmm1 {k1}, vm32z | A         | V/V                          | AVX512F                  | Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.128.66.0F38.W1 90 /vsib<br>VPGATHERDQ xmm1 {k1}, vm32x | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking. |
| EVEX.256.66.0F38.W1 90 /vsib<br>VPGATHERDQ ymm1 {k1}, vm32x | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking. |
| EVEX.512.66.0F38.W1 90 /vsib<br>VPGATHERDQ zmm1 {k1}, vm32y | A         | V/V                          | AVX512F                  | Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2   | Operand 3 | Operand 4 |
|-------|---------------|---------------|---|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | N/A       | N/A       |

### 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  $\text{disp8} * N$  and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

### Operation

BASE\_ADDR stands for the memory operand base address (a GPR); may not exist

VINDEX stands for the memory operand vector of indices (a ZMM register)

SCALE stands for the memory operand scalar (1, 2, 4 or 8)

DISP is the optional 1 or 4 byte displacement

### VPGATHERDD (EVEX encoded version)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

  i := j \* 32

  IF k1[j]

    THEN DEST[i+31:i] := MEM[BASE\_ADDR +  
      SignExtend(VINDEX[i+31:i]) \* SCALE + DISP]

    k1[j] := 0

    ELSE \*DEST[i+31:i] := remains unchanged\*       ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] := 0

DEST[MAXVL-1:VL] := 0

### VPGATHERDQ (EVEX encoded version)

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

  i := j \* 64

  k := j \* 32

  IF k1[j]

    THEN DEST[i+63:i] :=  
      MEM[BASE\_ADDR + SignExtend(VINDEX[k+31:k]) \* SCALE + DISP]

    k1[j] := 0

    ELSE \*DEST[i+63:i] := remains unchanged\*       ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] := 0

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPGATHERDD __m512i __mm512_i32gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERDD __m512i __mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i __mm256_mask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i __mm_mask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i __mm512_i32logather_epi64(__m256i vdx, void * base, int scale);
VPGATHERDQ __m512i __mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDQ __m256i __mm256_mask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i __mm_mask_i32gather_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

| Opcode/<br>Instruction                                   | Op/<br>En | 64/32<br>-bit<br>Mode | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|-----------------------|--------------------------|--|
| VEX.128.66.0F38.W1 90 /r<br>VPGATHERDQ xmm1, vm32x, xmm2 | A         | V/V                   | AVX2                     | Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |
| VEX.128.66.0F38.W1 91 /r<br>VPGATHERQQ xmm1, vm64x, xmm2 | A         | V/V                   | AVX2                     | Using qword indices specified in vm64x, gather qword values from memory conditioned on mask specified by xmm2. Conditionally gathered elements are merged into xmm1. |
| VEX.256.66.0F38.W1 90 /r<br>VPGATHERDQ ymm1, vm32x, ymm2 | A         | V/V                   | AVX2                     | Using dword indices specified in vm32x, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1. |
| VEX.256.66.0F38.W1 91 /r<br>VPGATHERQQ ymm1, vm64y, ymm2 | A         | V/V                   | AVX2                     | Using qword indices specified in vm64y, gather qword values from memory conditioned on mask specified by ymm2. Conditionally gathered elements are merged into ymm1. |

### Instruction Operand Encoding

| Op/En | Operand 1       | Operand 2   | Operand 3       | Operand 4 |
|-------|-----------------|---|-----------------|-----------|
| A     | ModRM:reg (r,w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | VEX.vvvv (r, w) | N/A       |

### Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST := SRC1;  
 BASE\_ADDR: base register encoded in VSIB addressing;  
 VINDEXT: 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[j +63:i] := FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[j +63:i] := 0;
  FI;
ENDFOR
FOR j := 0 to 1
  k := j * 32;
  i := j * 64;
  DATA_ADDR := BASE_ADDR + (SignExtend(VINDEXT[k+31:k])*SCALE + DISP);
  IF MASK[63+i] THEN
    DEST[j +63:i] := FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[j +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."

## VPATHERQD/VPATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

| Opcode/<br>Instruction                                     | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 91 /vsib<br>VPATHERQD xmm1 {k1}, vm64x | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.256.66.0F38.W0 91 /vsib<br>VPATHERQD xmm1 {k1}, vm64y | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.512.66.0F38.W0 91 /vsib<br>VPATHERQD ymm1 {k1}, vm64z | A         | V/V                          | AVX512F                  | Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.    |
| EVEX.128.66.0F38.W1 91 /vsib<br>VPATHERQQ xmm1 {k1}, vm64x | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking. |
| EVEX.256.66.0F38.W1 91 /vsib<br>VPATHERQQ ymm1 {k1}, vm64y | A         | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking. |
| EVEX.512.66.0F38.W1 91 /vsib<br>VPATHERQQ zmm1 {k1}, vm64z | A         | V/V                          | AVX512F                  | Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2   | Operand 3 | Operand 4 |
|-------|---------------|---------------|---|-----------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | N/A       | N/A       |

### 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  $\text{disp}8*N$  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]

    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.”

## VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 44 /r<br>VPLZCNTD xmm1 {k1}{z},<br>xmm2/m128/m32bcst | A         | V/V                          | AVX512VL<br>AVX512CD     | Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1. |
| EVEX.256.66.0F38.W0 44 /r<br>VPLZCNTD ymm1 {k1}{z},<br>ymm2/m256/m32bcst | A         | V/V                          | AVX512VL<br>AVX512CD     | Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1. |
| EVEX.512.66.0F38.W0 44 /r<br>VPLZCNTD zmm1 {k1}{z},<br>zmm2/m512/m32bcst | A         | V/V                          | AVX512CD                 | Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1. |
| EVEX.128.66.0F38.W1 44 /r<br>VPLZCNTQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst | A         | V/V                          | AVX512VL<br>AVX512CD     | Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1. |
| EVEX.256.66.0F38.W1 44 /r<br>VPLZCNTQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst | A         | V/V                          | AVX512VL<br>AVX512CD     | Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1. |
| EVEX.512.66.0F38.W1 44 /r<br>VPLZCNTQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst | A         | V/V                          | AVX512CD                 | Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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 &gt; 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 &gt; 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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID                   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.66.0F38.W1 B5 /r<br>VPMADD52HUQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | A         | V/V                          | AVX512_IFMA<br>AVX512VL | Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 B5 /r<br>VPMADD52HUQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | A         | V/V                          | AVX512_IFMA<br>AVX512VL | 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. |
| EVEX.512.66.0F38.W1 B5 /r<br>VPMADD52HUQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | A         | V/V                          | AVX512_IFMA             | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3    | Operand 4 |
|-------|------------|------------------|---------------|--------------|-----------|
| A     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m(r) | N/A       |

### Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The high 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.



**Operation****VPMADD52HUQ (EVEX encoded)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j \* 64;

IF k1[j] OR \*no writemask\* THEN

IF src2 is Memory AND EVEX.b=1 THEN

tsrc2[63:0] := ZeroExtend64(src2[51:0]);

ELSE

tsrc2[63:0] := ZeroExtend64(src2[i+51:i]);

FI;

Temp128[127:0] := ZeroExtend64(src1[i+51:i]) \* tsrc2[63:0];

Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[103:52]);

DEST[i+63:i] := Temp2[63:0];

ELSE

IF \*zeroing-masking\* THEN

DEST[i+63:i] := 0;

ELSE \*merge-masking\*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMADD52HUQ \_\_m512i \_\_mm512\_madd52hi\_epu64( \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m512i \_\_mm512\_mask\_madd52hi\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m512i \_\_mm512\_maskz\_madd52hi\_epu64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_madd52hi\_epu64( \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_mask\_madd52hi\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m256i \_\_mm256\_maskz\_madd52hi\_epu64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52HUQ \_\_m128i \_\_mm\_madd52hi\_epu64( \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52HUQ \_\_m128i \_\_mm\_mask\_madd52hi\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52HUQ \_\_m128i \_\_mm\_maskz\_madd52hi\_epu64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See 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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>Bit Mode<br>Support | CPUID                   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.0F38.W1 B4 /r<br>VPMADD52LUQ xmm1 {k1}{z},<br>xmm2,xmm3/m128/m64bcst  | A         | V/V                          | AVX512_IFMA<br>AVX512VL | Multiply unsigned 52-bit integers in xmm2 and xmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in xmm1 using writemask k1. |
| EVEX.256.66.0F38.W1 B4 /r<br>VPMADD52LUQ ymm1 {k1}{z},<br>ymm2, ymm3/m256/m64bcst | A         | V/V                          | AVX512_IFMA<br>AVX512VL | 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. |
| EVEX.512.66.0F38.W1 B4 /r<br>VPMADD52LUQ zmm1 {k1}{z},<br>zmm2,zmm3/m512/m64bcst  | A         | V/V                          | AVX512_IFMA             | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3    | Operand 4 |
|-------|------------|------------------|---------------|--------------|-----------|
| A     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m(r) | N/A       |

### Description

Multiplies packed unsigned 52-bit integers in each qword element of the first source operand (the second operand) with the packed unsigned 52-bit integers in the corresponding elements of the second source operand (the third operand) to form packed 104-bit intermediate results. The low 52-bit, unsigned integer of each 104-bit product is added to the corresponding qword unsigned integer of the destination operand (the first operand) under the writemask k1.

The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 64-bit granularity.

**Operation****VPMADD52LUQ (EVEX encoded)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j \* 64;

IF k1[j] OR \*no writemask\* THEN

IF src2 is Memory AND EVEX.b=1 THEN

tsrc2[63:0] := ZeroExtend64(src2[51:0]);

ELSE

tsrc2[63:0] := ZeroExtend64(src2[i+51:i]);

FI;

Temp128[127:0] := ZeroExtend64(src1[i+51:i]) \* tsrc2[63:0];

Temp2[63:0] := DEST[i+63:i] + ZeroExtend64(temp128[51:0]);

DEST[i+63:i] := Temp2[63:0];

ELSE

IF \*zeroing-masking\* THEN

DEST[i+63:i] := 0;

ELSE \*merge-masking\*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] := 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMADD52LUQ \_\_m512i \_\_mm512\_madd52lo\_epu64( \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m512i \_\_mm512\_mask\_madd52lo\_epu64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m512i \_\_mm512\_maskz\_madd52lo\_epu64( \_\_mmask8 k, \_\_m512i a, \_\_m512i b, \_\_m512i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_madd52lo\_epu64( \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_mask\_madd52lo\_epu64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m256i \_\_mm256\_maskz\_madd52lo\_epu64( \_\_mmask8 k, \_\_m256i a, \_\_m256i b, \_\_m256i c);

VPMADD52LUQ \_\_m128i \_\_mm\_madd52lo\_epu64( \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52LUQ \_\_m128i \_\_mm\_mask\_madd52lo\_epu64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

VPMADD52LUQ \_\_m128i \_\_mm\_maskz\_madd52lo\_epu64( \_\_mmask8 k, \_\_m128i a, \_\_m128i b, \_\_m128i c);

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions."

## VPMASKMOV—Conditional SIMD Integer Packed Loads and Stores

| Opcode/<br>Instruction                                  | Op/<br>En | 64/32<br>-bit<br>Mode | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|-----------------------|--------------------------|---|
| VEX.128.66.0F38.W0 8C /r<br>VPMASKMOVD xmm1, xmm2, m128 | RVM       | V/V                   | AVX2                     | Conditionally load dword values from m128 using mask in xmm2 and store in xmm1. |
| VEX.256.66.0F38.W0 8C /r<br>VPMASKMOVD ymm1, ymm2, m256 | RVM       | V/V                   | AVX2                     | Conditionally load dword values from m256 using mask in ymm2 and store in ymm1. |
| VEX.128.66.0F38.W1 8C /r<br>VPMASKMOVQ xmm1, xmm2, m128 | RVM       | V/V                   | AVX2                     | Conditionally load qword values from m128 using mask in xmm2 and store in xmm1. |
| VEX.256.66.0F38.W1 8C /r<br>VPMASKMOVQ ymm1, ymm2, m256 | RVM       | V/V                   | AVX2                     | Conditionally load qword values from m256 using mask in ymm2 and store in ymm1. |
| VEX.128.66.0F38.W0 8E /r<br>VPMASKMOVD m128, xmm1, xmm2 | MVR       | V/V                   | AVX2                     | Conditionally store dword values from xmm2 using mask in xmm1.                  |
| VEX.256.66.0F38.W0 8E /r<br>VPMASKMOVD m256, ymm1, ymm2 | MVR       | V/V                   | AVX2                     | Conditionally store dword values from ymm2 using mask in ymm1.                  |
| VEX.128.66.0F38.W1 8E /r<br>VPMASKMOVQ m128, xmm1, xmm2 | MVR       | V/V                   | AVX2                     | Conditionally store qword values from xmm2 using mask in xmm1.                  |
| VEX.256.66.0F38.W1 8E /r<br>VPMASKMOVQ m256, ymm1, ymm2 | MVR       | V/V                   | AVX2                     | Conditionally store qword values from ymm2 using mask in ymm1.                  |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|---------------|--------------|---------------|-----------|
| RVM   | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| MVR   | ModRM:r/m (w) | VEX.vvvv (r) | ModRM:reg (r) | N/A       |

### 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.

VPMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm\_field, and the destination register is encoded in reg\_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg\_field, and the destination memory location is encoded in rm\_field.

## Operation

### VPMASKMOVD - 256-bit load

```
DEST[31:0] := IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] := IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] := IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] := IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] := IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] := IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] := IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] := IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

### VPMASKMOVD -128-bit load

```
DEST[31:0] := IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] := IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] := IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] := IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[MAXVL-1:128] := 0
```

### VPMASKMOVQ - 256-bit load

```
DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] := IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] := IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

### VPMASKMOVQ - 128-bit load

```
DEST[63:0] := IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] := IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[MAXVL-1:128] := 0
```

### VPMASKMOVD - 256-bit store

```
IF (SRC1[31]) DEST[31:0] := SRC2[31:0]
IF (SRC1[63]) DEST[63:32] := SRC2[63:32]
IF (SRC1[95]) DEST[95:64] := SRC2[95:64]
IF (SRC1[127]) DEST[127:96] := SRC2[127:96]
IF (SRC1[159]) DEST[159:128] := SRC2[159:128]
IF (SRC1[191]) DEST[191:160] := SRC2[191:160]
IF (SRC1[223]) DEST[223:192] := SRC2[223:192]
IF (SRC1[255]) DEST[255:224] := SRC2[255:224]
```

**VPMASKMOVD - 128-bit store**

IF (SRC1[31]) DEST[31:0] := SRC2[31:0]  
 IF (SRC1[63]) DEST[63:32] := SRC2[63:32]  
 IF (SRC1[95]) DEST[95:64] := SRC2[95:64]  
 IF (SRC1[127]) DEST[127:96] := SRC2[127:96]

**VPMASKMOVQ - 256-bit store**

IF (SRC1[63]) DEST[63:0] := SRC2[63:0]  
 IF (SRC1[127]) DEST[127:64] := SRC2[127:64]  
 IF (SRC1[191]) DEST[191:128] := SRC2[191:128]  
 IF (SRC1[255]) DEST[255:192] := SRC2[255:192]

**VPMASKMOVQ - 128-bit store**

IF (SRC1[63]) DEST[63:0] := SRC2[63:0]  
 IF (SRC1[127]) DEST[127:64] := SRC2[127:64]

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMASKMOVD: `__m256i _mm256_maskload_epi32(int const *a, __m256i mask)`  
 VPMASKMOVD: `void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)`  
 VPMASKMOVQ: `__m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);`  
 VPMASKMOVQ: `void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);`  
 VPMASKMOVD: `__m128i _mm_maskload_epi32(int const *a, __m128i mask)`  
 VPMASKMOVD: `void _mm_maskstore_epi32(int *a, __m128i mask, __m128 b)`  
 VPMASKMOVQ: `__m128i _mm_maskload_epi64(__int64 const *a, __m128i mask);`  
 VPMASKMOVQ: `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/<br>Instruction                         | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 29 /r<br>VPMOVB2M k1, xmm1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.       |
| EVEX.256.F3.0F38.W0 29 /r<br>VPMOVB2M k1, ymm1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.       |
| EVEX.512.F3.0F38.W0 29 /r<br>VPMOVB2M k1, zmm1 | RM        | V/V                          | AVX512BW                 | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.       |
| EVEX.128.F3.0F38.W1 29 /r<br>VPMOVW2M k1, xmm1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.       |
| EVEX.256.F3.0F38.W1 29 /r<br>VPMOVW2M k1, ymm1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.       |
| EVEX.512.F3.0F38.W1 29 /r<br>VPMOVW2M k1, zmm1 | RM        | V/V                          | AVX512BW                 | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.       |
| EVEX.128.F3.0F38.W0 39 /r<br>VPMOVD2M k1, xmm1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1. |
| EVEX.256.F3.0F38.W0 39 /r<br>VPMOVD2M k1, ymm1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1. |
| EVEX.512.F3.0F38.W0 39 /r<br>VPMOVD2M k1, zmm1 | RM        | V/V                          | AVX512DQ                 | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1. |
| EVEX.128.F3.0F38.W1 39 /r<br>VPMOVQ2M k1, xmm1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.   |
| EVEX.256.F3.0F38.W1 39 /r<br>VPMOVQ2M k1, ymm1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.   |
| EVEX.512.F3.0F38.W1 39 /r<br>VPMOVQ2M k1, zmm1 | RM        | V/V                          | AVX512DQ                 | Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.   |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

#### Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVB2M (EVEX encoded versions)**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1

i := j \* 8

IF SRC[i+7]

THEN DEST[j] := 1

ELSE DEST[j] := 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

**VPMOVW2M (EVEX encoded versions)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1

i := j \* 16

IF SRC[i+15]

THEN DEST[j] := 1

ELSE DEST[j] := 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

**VPMOVD2M (EVEX encoded versions)**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

i := j \* 32

IF SRC[i+31]

THEN DEST[j] := 1

ELSE DEST[j] := 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0

**VPMOVQ2M (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j \* 64

IF SRC[i+63]

THEN DEST[j] := 1

ELSE DEST[j] := 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] := 0



**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMPPOVB2M __mmask64 _mm512_movepi8_mask( __m512i );
VPMPPOVD2M __mmask16 _mm512_movepi32_mask( __m512i );
VPMPPOVQ2M __mmask8 _mm512_movepi64_mask( __m512i );
VPMPPOVW2M __mmask32 _mm512_movepi16_mask( __m512i );
VPMPPOVB2M __mmask32 _mm256_movepi8_mask( __m256i );
VPMPPOVD2M __mmask8 _mm256_movepi32_mask( __m256i );
VPMPPOVQ2M __mmask8 _mm256_movepi64_mask( __m256i );
VPMPPOVW2M __mmask16 _mm256_movepi16_mask( __m256i );
VPMPPOVB2M __mmask16 _mm_movepi8_mask( __m128i );
VPMPPOVD2M __mmask8 _mm_movepi32_mask( __m128i );
VPMPPOVQ2M __mmask8 _mm_movepi64_mask( __m128i );
VPMPPOVW2M __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.

## VPMOVD/VPMSDB/VPMUSDB—Down Convert DWord to Byte

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 31 /r<br>VPMOVD xmm1/m32 {k1}{z}, xmm2      | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed double-word integers from xmm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.                                |
| EVEX.128.F3.0F38.W0 21 /r<br>VPMSDB xmm1/m32 {k1}{z}, xmm2      | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed signed double-word integers from xmm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.          |
| EVEX.128.F3.0F38.W0 11 /r<br>VPMUSDB xmm1/m32 {k1}{z}, xmm2     | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1.    |
| EVEX.256.F3.0F38.W0 31 /r<br>VPMOVD xmm1/m64 {k1}{z}, ymm2      | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed double-word integers from ymm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.                                |
| EVEX.256.F3.0F38.W0 21 /r<br>VPMSDB xmm1/m64 {k1}{z}, ymm2      | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed signed double-word integers from ymm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.          |
| EVEX.256.F3.0F38.W0 11 /r<br>VPMUSDB xmm1/m64 {k1}{z}, ymm2     | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1.    |
| EVEX.512.F3.0F38.W0 31 /r<br>VPMOVD xmm1/m128 {k1}{z}, zmm2     | A          | V/V                          | AVX512F                  | Converts 16 packed double-word integers from zmm2 into 16 packed byte integers in xmm1/m128 with truncation under writemask k1.                             |
| EVEX.512.F3.0F38.W0 21 /r<br>VPMSDB xmm1/m128 {k1}{z}, zmm2     | A          | V/V                          | AVX512F                  | Converts 16 packed signed double-word integers from zmm2 into 16 packed signed byte integers in xmm1/m128 using signed saturation under writemask k1.       |
| EVEX.512.F3.0F38.W0 11 /r<br>VPMUSDB xmm1/m128 {k1}{z},<br>zmm2 | A          | V/V                          | AVX512F                  | Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned byte integers in xmm1/m128 using unsigned saturation under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------------|---------------|---------------|-----------|-----------|
| A     | Quarter Mem | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

#### Description

VPMOVD down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMUSDB convert unsigned double-word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

### Operation

#### VPMOVDDB 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;
```

#### VPMOVDDB 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
```

#### VPMOVSDB 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;
```

**VPMOVSDB 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

DEST[MAXVL-1:VL/4] := 0;

**VPMOVUSDB instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

i := j \* 8

m := j \* 32

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] := SaturateUnsignedDoubleWordToByte (SRC[m+31:m])

ELSE \*DEST[i+7:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

### Intel C/C++ Compiler Intrinsic Equivalents

```

VPMOVBDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVBDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVBDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVBDB void _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVSDB __m128i _mm512_cvtsepi32_epi8(__m512i a);
VPMOVSDB __m128i _mm512_mask_cvtsepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVSDB __m128i _mm512_maskz_cvtsepi32_epi8(__mmask16 k, __m512i a);
VPMOVSDB void _mm512_mask_cvtsepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_cvtusepi32_epi8(__m512i a);
VPMOVUSDB __m128i _mm512_mask_cvtusepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_maskz_cvtusepi32_epi8(__mmask16 k, __m512i a);
VPMOVUSDB void _mm512_mask_cvtusepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm256_cvtusepi32_epi8(__m256i a);
VPMOVUSDB __m128i _mm256_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm256_maskz_cvtusepi32_epi8(__mmask8 k, __m256i b);
VPMOVUSDB void _mm256_mask_cvtusepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm_cvtusepi32_epi8(__m128i a);
VPMOVUSDB __m128i _mm_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDB __m128i _mm_maskz_cvtusepi32_epi8(__mmask8 k, __m128i b);
VPMOVUSDB void _mm_mask_cvtusepi32_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVSDB __m128i _mm256_cvtsepi32_epi8(__m256i a);
VPMOVSDB __m128i _mm256_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVSDB __m128i _mm256_maskz_cvtsepi32_epi8(__mmask8 k, __m256i b);
VPMOVSDB void _mm256_mask_cvtsepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVSDB __m128i _mm_cvtsepi32_epi8(__m128i a);
VPMOVSDB __m128i _mm_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSDB __m128i _mm_maskz_cvtsepi32_epi8(__mmask8 k, __m128i b);
VPMOVSDB void _mm_mask_cvtsepi32_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVBDB __m128i _mm256_cvtepi32_epi8(__m256i a);
VPMOVBDB __m128i _mm256_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVBDB __m128i _mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i b);
VPMOVBDB void _mm256_mask_cvtepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVBDB __m128i _mm_cvtepi32_epi8(__m128i a);
VPMOVBDB __m128i _mm_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVBDB __m128i _mm_maskz_cvtepi32_epi8(__mmask8 k, __m128i b);
VPMOVBDB void _mm_mask_cvtepi32_storeu_epi8(void *, __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.

## VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 33 /r<br>VPMOVDW xmm1/m64 {k1}{z}, xmm2       | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed double-word integers from xmm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.                                |
| EVEX.128.F3.0F38.W0 23 /r<br>VPMOVSDW xmm1/m64 {k1}{z}, xmm2      | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed signed double-word integers from xmm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.          |
| EVEX.128.F3.0F38.W0 13 /r<br>VPMOVUSDW xmm1/m64 {k1}{z}, xmm2     | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed unsigned double-word integers from xmm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.    |
| EVEX.256.F3.0F38.W0 33 /r<br>VPMOVDW xmm1/m128 {k1}{z}, ymm2      | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed double-word integers from ymm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.                               |
| EVEX.256.F3.0F38.W0 23 /r<br>VPMOVSDW xmm1/m128 {k1}{z}, ymm2     | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed signed double-word integers from ymm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.         |
| EVEX.256.F3.0F38.W0 13 /r<br>VPMOVUSDW xmm1/m128 {k1}{z},<br>ymm2 | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed unsigned double-word integers from ymm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1.   |
| EVEX.512.F3.0F38.W0 33 /r<br>VPMOVDW ymm1/m256 {k1}{z}, zmm2      | A          | V/V                          | AVX512F                  | Converts 16 packed double-word integers from zmm2 into 16 packed word integers in ymm1/m256 with truncation under writemask k1.                             |
| EVEX.512.F3.0F38.W0 23 /r<br>VPMOVSDW ymm1/m256 {k1}{z}, zmm2     | A          | V/V                          | AVX512F                  | Converts 16 packed signed double-word integers from zmm2 into 16 packed signed word integers in ymm1/m256 using signed saturation under writemask k1.       |
| EVEX.512.F3.0F38.W0 13 /r<br>VPMOVUSDW ymm1/m256 {k1}{z},<br>zmm2 | A          | V/V                          | AVX512F                  | Converts 16 packed unsigned double-word integers from zmm2 into 16 packed unsigned word integers in ymm1/m256 using unsigned saturation under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Half Mem   | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

### 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
```

#### VPMOVS DW 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;
```

**VPMOVS<sub>DW</sub> 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

```

**VPMOV<sub>USDW</sub> 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;

```

**VPMOV<sub>USDW</sub> instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```



### Intel C/C++ Compiler Intrinsic Equivalents

```

VPMOVDW __m256i __mm512_cvtepi32_epi16(__m512i a);
VPMOVDW __m256i __mm512_mask_cvtepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i __mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void __mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVSDW __m256i __mm512_cvtsepi32_epi16(__m512i a);
VPMOVSDW __m256i __mm512_mask_cvtsepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVSDW __m256i __mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
VPMOVSDW void __mm512_mask_cvtsepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i __mm512_cvtusepi32_epi16(__m512i a);
VPMOVUSDW __m256i __mm512_mask_cvtusepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i __mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void __mm512_mask_cvtusepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m128i __mm256_cvtusepi32_epi16(__m256i a);
VPMOVUSDW __m128i __mm256_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDW __m128i __mm256_maskz_cvtusepi32_epi16(__mmask8 k, __m256i b);
VPMOVUSDW void __mm256_mask_cvtusepi32_storeu_epi16(void *, __mmask8 k, __m256i b);
VPMOVUSDW __m128i __mm_cvtusepi32_epi16(__m128i a);
VPMOVUSDW __m128i __mm_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDW __m128i __mm_maskz_cvtusepi32_epi16(__mmask8 k, __m128i b);
VPMOVUSDW void __mm_mask_cvtusepi32_storeu_epi16(void *, __mmask8 k, __m128i b);
VPMOVSDW __m128i __mm256_cvtsepi32_epi16(__m256i a);
VPMOVSDW __m128i __mm256_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVSDW __m128i __mm256_maskz_cvtsepi32_epi16(__mmask8 k, __m256i b);
VPMOVSDW void __mm256_mask_cvtsepi32_storeu_epi16(void *, __mmask8 k, __m256i b);
VPMOVSDW __m128i __mm_cvtsepi32_epi16(__m128i a);
VPMOVSDW __m128i __mm_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVSDW __m128i __mm_maskz_cvtsepi32_epi16(__mmask8 k, __m128i b);
VPMOVSDW void __mm_mask_cvtsepi32_storeu_epi16(void *, __mmask8 k, __m128i b);
VPMOVDW __m128i __mm256_cvtepi32_epi16(__m256i a);
VPMOVDW __m128i __mm256_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm256_maskz_cvtepi32_epi16(__mmask8 k, __m256i b);
VPMOVDW void __mm256_mask_cvtepi32_storeu_epi16(void *, __mmask8 k, __m256i b);
VPMOVDW __m128i __mm_cvtepi32_epi16(__m128i a);
VPMOVDW __m128i __mm_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVDW __m128i __mm_maskz_cvtepi32_epi16(__mmask8 k, __m128i b);
VPMOVDW void __mm_mask_cvtepi32_storeu_epi16(void *, __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.

## VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

| Opcode/<br>Instruction                         | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 28 /r<br>VPMOVM2B xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.256.F3.0F38.W0 28 /r<br>VPMOVM2B ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.512.F3.0F38.W0 28 /r<br>VPMOVM2B zmm1, k1 | RM        | V/V                          | AVX512BW                 | Sets each byte in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.128.F3.0F38.W1 28 /r<br>VPMOVM2W xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.256.F3.0F38.W1 28 /r<br>VPMOVM2W ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512BW     | Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.512.F3.0F38.W1 28 /r<br>VPMOVM2W zmm1, k1 | RM        | V/V                          | AVX512BW                 | Sets each word in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.       |
| EVEX.128.F3.0F38.W0 38 /r<br>VPMOVM2D xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1. |
| EVEX.256.F3.0F38.W0 38 /r<br>VPMOVM2D ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1. |
| EVEX.512.F3.0F38.W0 38 /r<br>VPMOVM2D zmm1, k1 | RM        | V/V                          | AVX512DQ                 | Sets each doubleword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1. |
| EVEX.128.F3.0F38.W1 38 /r<br>VPMOVM2Q xmm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.   |
| EVEX.256.F3.0F38.W1 38 /r<br>VPMOVM2Q ymm1, k1 | RM        | V/V                          | AVX512VL<br>AVX512DQ     | Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.   |
| EVEX.512.F3.0F38.W1 38 /r<br>VPMOVM2Q zmm1, k1 | RM        | V/V                          | AVX512DQ                 | Sets each quadword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.   |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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:i] := -1

ELSE DEST[i+7:i] := 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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.F3.0F38.W0 32 /r<br>VPMOVQB xmm1/m16 {k1}{z}, xmm2   | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed quad-word integers from xmm2 into 2 packed byte integers in xmm1/m16 with truncation under writemask k1.                             |
| EVEX.128.F3.0F38.W0 22 /r<br>VPMOVSQB xmm1/m16 {k1}{z}, xmm2  | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed signed quad-word integers from xmm2 into 2 packed signed byte integers in xmm1/m16 using signed saturation under writemask k1.       |
| EVEX.128.F3.0F38.W0 12 /r<br>VPMOVUSQB xmm1/m16 {k1}{z}, xmm2 | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned byte integers in xmm1/m16 using unsigned saturation under writemask k1. |
| EVEX.256.F3.0F38.W0 32 /r<br>VPMOVQB xmm1/m32 {k1}{z}, ymm2   | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed quad-word integers from ymm2 into 4 packed byte integers in xmm1/m32 with truncation under writemask k1.                             |
| EVEX.256.F3.0F38.W0 22 /r<br>VPMOVSQB xmm1/m32 {k1}{z}, ymm2  | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed byte integers in xmm1/m32 using signed saturation under writemask k1.       |
| EVEX.256.F3.0F38.W0 12 /r<br>VPMOVUSQB xmm1/m32 {k1}{z}, ymm2 | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned byte integers in xmm1/m32 using unsigned saturation under writemask k1. |
| EVEX.512.F3.0F38.W0 32 /r<br>VPMOVQB xmm1/m64 {k1}{z}, zmm2   | A          | V/V                          | AVX512F                  | Converts 8 packed quad-word integers from zmm2 into 8 packed byte integers in xmm1/m64 with truncation under writemask k1.                             |
| EVEX.512.F3.0F38.W0 22 /r<br>VPMOVSQB xmm1/m64 {k1}{z}, zmm2  | A          | V/V                          | AVX512F                  | Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed byte integers in xmm1/m64 using signed saturation under writemask k1.       |
| EVEX.512.F3.0F38.W0 12 /r<br>VPMOVUSQB xmm1/m64 {k1}{z}, zmm2 | A          | V/V                          | AVX512F                  | Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned byte integers in xmm1/m64 using unsigned saturation under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Eighth Mem | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

#### Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:64) of the destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

**Operation****VPMOVQB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 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
  FI;
ENDFOR
DEST[MAXVL-1:VL/8] := 0;

```

**VPMOVQB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 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

```

**VPMOVSQB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 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
  FI;
ENDFOR
DEST[MAXVL-1:VL/8] := 0;

```

**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**

```

VPMOVBQ __m128i __mm512_cvtepi64_epi8( __m512i a);
VPMOVBQ __m128i __mm512_mask_cvtepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVBQ void __mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_cvtsepi64_epi8( __m512i a);
VPMOVBQ __m128i __mm512_mask_cvtsepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_maskz_cvtsepi64_epi8( __mmask8 k, __m512i a);
VPMOVBQ void __mm512_mask_cvtsepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_cvtusepi64_epi8( __m512i a);
VPMOVBQ __m128i __mm512_mask_cvtusepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm512_maskz_cvtusepi64_epi8( __mmask8 k, __m512i a);
VPMOVBQ void __mm512_mask_cvtusepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVBQ __m128i __mm256_cvtusepi64_epi8(__m256i a);
VPMOVBQ __m128i __mm256_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm256_maskz_cvtusepi64_epi8( __mmask8 k, __m256i b);
VPMOVBQ void __mm256_mask_cvtusepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm_cvtusepi64_epi8(__m128i a);
VPMOVBQ __m128i __mm_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm_maskz_cvtusepi64_epi8( __mmask8 k, __m128i b);
VPMOVBQ void __mm_mask_cvtusepi64_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm256_cvtsepi64_epi8(__m256i a);
VPMOVBQ __m128i __mm256_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm256_maskz_cvtsepi64_epi8( __mmask8 k, __m256i b);
VPMOVBQ void __mm256_mask_cvtsepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm_cvtsepi64_epi8(__m128i a);
VPMOVBQ __m128i __mm_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm_maskz_cvtsepi64_epi8( __mmask8 k, __m128i b);
VPMOVBQ void __mm_mask_cvtsepi64_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm256_cvtepi64_epi8(__m256i a);
VPMOVBQ __m128i __mm256_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm256_maskz_cvtepi64_epi8( __mmask8 k, __m256i b);
VPMOVBQ void __mm256_mask_cvtepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVBQ __m128i __mm_cvtepi64_epi8(__m128i a);
VPMOVBQ __m128i __mm_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVBQ __m128i __mm_maskz_cvtepi64_epi8( __mmask8 k, __m128i b);
VPMOVBQ void __mm_mask_cvtepi64_storeu_epi8(void *, __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.



## VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

| Opcode/<br>Instruction   | Op /<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|-----------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 35 /r<br>VPMOVQD xmm1/m128 {k1}{z}, xmm2   | A          | V/V                         | AVX512VL<br>AVX512F      | Converts 2 packed quad-word integers from xmm2 into 2 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.                             |
| EVEX.128.F3.0F38.W0 25 /r<br>VPMOVSQD xmm1/m64 {k1}{z}, xmm2   | A          | V/V                         | AVX512VL<br>AVX512F      | 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.        |
| EVEX.128.F3.0F38.W0 15 /r<br>VPMOVUSQD xmm1/m64 {k1}{z}, xmm2  | A          | V/V                         | AVX512VL<br>AVX512F      | 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.  |
| EVEX.256.F3.0F38.W0 35 /r<br>VPMOVQD xmm1/m128 {k1}{z}, ymm2   | A          | V/V                         | AVX512VL<br>AVX512F      | Converts 4 packed quad-word integers from ymm2 into 4 packed double-word integers in xmm1/m128 with truncation subject to writemask k1.                             |
| EVEX.256.F3.0F38.W0 25 /r<br>VPMOVSQD xmm1/m128 {k1}{z}, ymm2  | A          | V/V                         | AVX512VL<br>AVX512F      | 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.       |
| EVEX.256.F3.0F38.W0 15 /r<br>VPMOVUSQD xmm1/m128 {k1}{z}, ymm2 | A          | V/V                         | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.F3.0F38.W0 35 /r<br>VPMOVQD ymm1/m256 {k1}{z}, zmm2   | A          | V/V                         | AVX512F                  | Converts 8 packed quad-word integers from zmm2 into 8 packed double-word integers in ymm1/m256 with truncation subject to writemask k1.                             |
| EVEX.512.F3.0F38.W0 25 /r<br>VPMOVSQD ymm1/m256 {k1}{z}, zmm2  | A          | V/V                         | AVX512F                  | 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.       |
| EVEX.512.F3.0F38.W0 15 /r<br>VPMOVUSQD ymm1/m256 {k1}{z}, zmm2 | A          | V/V                         | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Half Mem   | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

#### 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
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
```

#### VPMOVSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] 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;
```

**VPMOVSQD instruction (EVEX encoded version) memory form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SaturateSignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSQD instruction (EVEX encoded version) reg-reg form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE
      IF *merging-masking*      ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*  ; zeroing-masking
          DEST[i+31:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;

```

**VPMOVUSQD instruction (EVEX encoded version) memory form**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 32
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
    ELSE *DEST[i+31:i] remains unchanged*      ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQD __m256i __mm512_cvtepi64_epi32( __m512i a);
VPMOVQD __m256i __mm512_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVQD __m256i __mm512_maskz_cvtepi64_epi32( __mmask8 k, __m512i a);
VPMOVQD void __mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVSQD __m256i __mm512_cvtsepi64_epi32( __m512i a);
VPMOVSQD __m256i __mm512_mask_cvtsepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVSQD __m256i __mm512_maskz_cvtsepi64_epi32( __mmask8 k, __m512i a);
VPMOVSQD void __mm512_mask_cvtsepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m256i __mm512_cvtusepi64_epi32( __m512i a);
VPMOVUSQD __m256i __mm512_mask_cvtusepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVUSQD __m256i __mm512_maskz_cvtusepi64_epi32( __mmask8 k, __m512i a);
VPMOVUSQD void __mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i __mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i __mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i __mm256_maskz_cvtusepi64_epi32( __mmask8 k, __m256i b);
VPMOVUSQD void __mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i __mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i __mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i __mm_maskz_cvtusepi64_epi32( __mmask8 k, __m128i b);
VPMOVUSQD void __mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i __mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i __mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i __mm256_maskz_cvtsepi64_epi32( __mmask8 k, __m256i b);
VPMOVSQD void __mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i __mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i __mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i __mm_maskz_cvtsepi64_epi32( __mmask8 k, __m128i b);
VPMOVSQD void __mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i __mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i __mm256_maskz_cvtepi64_epi32( __mmask8 k, __m256i b);
VPMOVQD void __mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i __mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i __mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __mm_maskz_cvtepi64_epi32( __mmask8 k, __m128i b);
VPMOVQD void __mm_mask_cvtepi64_storeu_epi32(void * , __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

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.128.F3.0F38.W0 34 /r<br>VPMOVQW xmm1/m32 {k1}{z}, xmm2       | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed quad-word integers from xmm2 into 2 packed word integers in xmm1/m32 with truncation under writemask k1.                              |
| EVEX.128.F3.0F38.W0 24 /r<br>VPMOVSQW xmm1/m32 {k1}{z}, xmm2      | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m32 using signed saturation under writemask k1.        |
| EVEX.128.F3.0F38.W0 14 /r<br>VPMOVUSQW xmm1/m32 {k1}{z}, xmm2     | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 2 packed unsigned quad-word integers from xmm2 into 2 packed unsigned word integers in xmm1/m32 using unsigned saturation under writemask k1.  |
| EVEX.256.F3.0F38.W0 34 /r<br>VPMOVQW xmm1/m64 {k1}{z}, ymm2       | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed quad-word integers from ymm2 into 4 packed word integers in xmm1/m64 with truncation under writemask k1.                              |
| EVEX.256.F3.0F38.W0 24 /r<br>VPMOVSQW xmm1/m64 {k1}{z}, ymm2      | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed signed quad-word integers from ymm2 into 4 packed signed word integers in xmm1/m64 using signed saturation under writemask k1.        |
| EVEX.256.F3.0F38.W0 14 /r<br>VPMOVUSQW xmm1/m64 {k1}{z}, ymm2     | A          | V/V                          | AVX512VL<br>AVX512F      | Converts 4 packed unsigned quad-word integers from ymm2 into 4 packed unsigned word integers in xmm1/m64 using unsigned saturation under writemask k1.  |
| EVEX.512.F3.0F38.W0 34 /r<br>VPMOVQW xmm1/m128 {k1}{z}, zmm2      | A          | V/V                          | AVX512F                  | Converts 8 packed quad-word integers from zmm2 into 8 packed word integers in xmm1/m128 with truncation under writemask k1.                             |
| EVEX.512.F3.0F38.W0 24 /r<br>VPMOVSQW xmm1/m128 {k1}{z}, zmm2     | A          | V/V                          | AVX512F                  | Converts 8 packed signed quad-word integers from zmm2 into 8 packed signed word integers in xmm1/m128 using signed saturation under writemask k1.       |
| EVEX.512.F3.0F38.W0 14 /r<br>VPMOVUSQW xmm1/m128 {k1}{z},<br>zmm2 | A          | V/V                          | AVX512F                  | Converts 8 packed unsigned quad-word integers from zmm2 into 8 packed unsigned word integers in xmm1/m128 using unsigned saturation under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------------|---------------|---------------|-----------|-----------|
| A     | Quarter Mem | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

#### 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 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

#### VPMOVQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := TruncateQuadWordToWord (SRC[m+63:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR
```

#### VPMOVSQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;
```

**VPMOVSQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateSignedQuadWordToWord (SRC[m+63:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSQW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/4] := 0;

```

**VPMOVUSQW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 16
  m := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := SaturateUnsignedQuadWordToWord (SRC[m+63:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMOVQW __m128i __mm512_cvtepi64_epi16(__m512i a);
VPMOVQW __m128i __mm512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i __mm512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void __mm512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i __mm512_cvtsepi64_epi16(__m512i a);
VPMOVSQW __m128i __mm512_mask_cvtsepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i __mm512_maskz_cvtsepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW 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(__m128i 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);
VPMOVUSQD __m128i __mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i __mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i __mm256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVUSQD void __mm256_mask_cvtusepi64_storeu_epi32(void *, __mmask8 k, __m256i b);
VPMOVUSQD __m128i __mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i __mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i __mm_maskz_cvtusepi64_epi32(__mmask8 k, __m128i b);
VPMOVUSQD void __mm_mask_cvtusepi64_storeu_epi32(void *, __mmask8 k, __m128i b);
VPMOVSQD __m128i __mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i __mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i __mm256_maskz_cvtsepi64_epi32(__mmask8 k, __m256i b);
VPMOVSQD void __mm256_mask_cvtsepi64_storeu_epi32(void *, __mmask8 k, __m256i b);
VPMOVSQD __m128i __mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i __mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i __mm_maskz_cvtsepi64_epi32(__mmask8 k, __m128i b);
VPMOVSQD void __mm_mask_cvtsepi64_storeu_epi32(void *, __mmask8 k, __m128i b);
VPMOVQD __m128i __mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i __mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i __mm256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void __mm256_mask_cvtepi64_storeu_epi32(void *, __mmask8 k, __m256i b);
VPMOVQD __m128i __mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i __mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i __mm_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD void __mm_mask_cvtepi64_storeu_epi32(void *, __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.



## VPMOVB/WB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.F3.0F38.W0 30 /r<br>VPMOVBWB xmm1/m64 {k1}{z}, xmm2      | A          | V/V                          | AVX512VL<br>AVX512BW     | Converts 8 packed word integers from xmm2 into 8 packed bytes in xmm1/m64 with truncation under writemask k1.                                |
| EVEX.128.F3.0F38.W0 20 /r<br>VPMOVSWB xmm1/m64 {k1}{z},<br>xmm2   | A          | V/V                          | AVX512VL<br>AVX512BW     | Converts 8 packed signed word integers from xmm2 into 8 packed signed bytes in xmm1/m64 using signed saturation under writemask k1.          |
| EVEX.128.F3.0F38.W0 10 /r<br>VPMOVUSWB xmm1/m64 {k1}{z},<br>xmm2  | A          | V/V                          | AVX512VL<br>AVX512BW     | Converts 8 packed unsigned word integers from xmm2 into 8 packed unsigned bytes in xmm1/m64 using unsigned saturation under writemask k1.    |
| EVEX.256.F3.0F38.W0 30 /r<br>VPMOVBWB xmm1/m128 {k1}{z},<br>ymm2  | A          | V/V                          | AVX512VL<br>AVX512BW     | Converts 16 packed word integers from ymm2 into 16 packed bytes in xmm1/m128 with truncation under writemask k1.                             |
| EVEX.256.F3.0F38.W0 20 /r<br>VPMOVSWB xmm1/m128 {k1}{z},<br>ymm2  | A          | V/V                          | AVX512VL<br>AVX512BW     | Converts 16 packed signed word integers from ymm2 into 16 packed signed bytes in xmm1/m128 using signed saturation under writemask k1.       |
| EVEX.256.F3.0F38.W0 10 /r<br>VPMOVUSWB xmm1/m128 {k1}{z},<br>ymm2 | A          | V/V                          | AVX512VL<br>AVX512BW     | Converts 16 packed unsigned word integers from ymm2 into 16 packed unsigned bytes in xmm1/m128 using unsigned saturation under writemask k1. |
| EVEX.512.F3.0F38.W0 30 /r<br>VPMOVBWB ymm1/m256 {k1}{z},<br>zmm2  | A          | V/V                          | AVX512BW                 | Converts 32 packed word integers from zmm2 into 32 packed bytes in ymm1/m256 with truncation under writemask k1.                             |
| EVEX.512.F3.0F38.W0 20 /r<br>VPMOVSWB ymm1/m256 {k1}{z},<br>zmm2  | A          | V/V                          | AVX512BW                 | Converts 32 packed signed word integers from zmm2 into 32 packed signed bytes in ymm1/m256 using signed saturation under writemask k1.       |
| EVEX.512.F3.0F38.W0 10 /r<br>VPMOVUSWB ymm1/m256 {k1}{z},<br>zmm2 | A          | V/V                          | AVX512BW                 | Converts 32 packed unsigned word integers from zmm2 into 32 packed unsigned bytes in ymm1/m256 using unsigned saturation under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Half Mem   | ModRM:r/m (w) | ModRM:reg (r) | N/A       | N/A       |

#### Description

VPMOVBWB 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****VPMOVB instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KI-1

i := j \* 8

m := j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] := TruncateWordToByte (SRC[m+15:m])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] := 0;

**VPMOVB instruction (EVEX encoded versions) when dest is memory**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KI-1

i := j \* 8

m := j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] := TruncateWordToByte (SRC[m+15:m])

ELSE

\*DEST[i+7:i] remains unchanged\* ; merging-masking

FI;

ENDFOR

**VPMOVSWB instruction (EVEX encoded versions) when dest is a register**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KI-1

i := j \* 8

m := j \* 16

IF k1[j] OR \*no writemask\*

THEN DEST[i+7:i] := SaturateSignedWordToByte (SRC[m+15:m])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+7:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+7:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL/2] := 0;

**VPMOVS WB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KI-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

```

**VPMOVUS WB instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KI-1
  i := j * 8
  m := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+7:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+7:i] = 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] := 0;

```

**VPMOVUS WB instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j := 0 TO KI-1
  i := j * 8
  m := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+7:i] := SaturateUnsignedWordToByte (SRC[m+15:m])
  ELSE
    *DEST[i+7:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**Intel C/C++ Compiler Intrinsic Equivalents**

VPMOVUSWB \_\_m256i \_\_mm512\_cvtusepi16\_epi8(\_\_m512i a);  
 VPMOVUSWB \_\_m256i \_\_mm512\_mask\_cvtusepi16\_epi8(\_\_m256i a, \_\_mmask32 k, \_\_m512i b);  
 VPMOVUSWB \_\_m256i \_\_mm512\_maskz\_cvtusepi16\_epi8(\_\_mmask32 k, \_\_m512i b);  
 VPMOVUSWB void \_\_mm512\_mask\_cvtusepi16\_storeu\_epi8(void \*, \_\_mmask32 k, \_\_m512i b);  
 VPMOVSWB \_\_m256i \_\_mm512\_cvtsepi16\_epi8(\_\_m512i a);  
 VPMOVSWB \_\_m256i \_\_mm512\_mask\_cvtsepi16\_epi8(\_\_m256i a, \_\_mmask32 k, \_\_m512i b);  
 VPMOVSWB \_\_m256i \_\_mm512\_maskz\_cvtsepi16\_epi8(\_\_mmask32 k, \_\_m512i b);  
 VPMOVSWB void \_\_mm512\_mask\_cvtsepi16\_storeu\_epi8(void \*, \_\_mmask32 k, \_\_m512i b);  
 VPMOVWB \_\_m256i \_\_mm512\_cvtepi16\_epi8(\_\_m512i a);  
 VPMOVWB \_\_m256i \_\_mm512\_mask\_cvtepi16\_epi8(\_\_m256i a, \_\_mmask32 k, \_\_m512i b);  
 VPMOVWB \_\_m256i \_\_mm512\_maskz\_cvtepi16\_epi8(\_\_mmask32 k, \_\_m512i b);  
 VPMOVWB void \_\_mm512\_mask\_cvtepi16\_storeu\_epi8(void \*, \_\_mmask32 k, \_\_m512i b);  
 VPMOVUSWB \_\_m128i \_\_mm256\_cvtusepi16\_epi8(\_\_m256i a);  
 VPMOVUSWB \_\_m128i \_\_mm256\_mask\_cvtusepi16\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVUSWB \_\_m128i \_\_mm256\_maskz\_cvtusepi16\_epi8(\_\_mmask16 k, \_\_m256i b);  
 VPMOVUSWB void \_\_mm256\_mask\_cvtusepi16\_storeu\_epi8(void \*, \_\_mmask16 k, \_\_m256i b);  
 VPMOVUSWB \_\_m128i \_\_mm\_cvtusepi16\_epi8(\_\_m128i a);  
 VPMOVUSWB \_\_m128i \_\_mm\_mask\_cvtusepi16\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVUSWB \_\_m128i \_\_mm\_maskz\_cvtusepi16\_epi8(\_\_mmask8 k, \_\_m128i b);  
 VPMOVUSWB void \_\_mm\_mask\_cvtusepi16\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSWB \_\_m128i \_\_mm256\_cvtsepi16\_epi8(\_\_m256i a);  
 VPMOVSWB \_\_m128i \_\_mm256\_mask\_cvtsepi16\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSWB \_\_m128i \_\_mm256\_maskz\_cvtsepi16\_epi8(\_\_mmask16 k, \_\_m256i b);  
 VPMOVSWB void \_\_mm256\_mask\_cvtsepi16\_storeu\_epi8(void \*, \_\_mmask16 k, \_\_m256i b);  
 VPMOVSWB \_\_m128i \_\_mm\_cvtepi16\_epi8(\_\_m128i a);  
 VPMOVSWB \_\_m128i \_\_mm\_mask\_cvtepi16\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVSWB \_\_m128i \_\_mm\_maskz\_cvtepi16\_epi8(\_\_mmask8 k, \_\_m128i b);  
 VPMOVSWB void \_\_mm\_mask\_cvtepi16\_storeu\_epi8(void \*, \_\_mmask8 k, \_\_m128i b);  
 VPMOVWB \_\_m128i \_\_mm256\_cvtepi16\_epi8(\_\_m256i a);  
 VPMOVWB \_\_m128i \_\_mm256\_mask\_cvtepi16\_epi8(\_\_m128i a, \_\_mmask16 k, \_\_m256i b);  
 VPMOVWB \_\_m128i \_\_mm256\_maskz\_cvtepi16\_epi8(\_\_mmask16 k, \_\_m256i b);  
 VPMOVWB void \_\_mm256\_mask\_cvtepi16\_storeu\_epi8(void \*, \_\_mmask16 k, \_\_m256i b);  
 VPMOVWB \_\_m128i \_\_mm\_cvtepi16\_epi8(\_\_m128i a);  
 VPMOVWB \_\_m128i \_\_mm\_mask\_cvtepi16\_epi8(\_\_m128i a, \_\_mmask8 k, \_\_m128i b);  
 VPMOVWB \_\_m128i \_\_mm\_maskz\_cvtepi16\_epi8(\_\_mmask8 k, \_\_m128i b);  
 VPMOVWB void \_\_mm\_mask\_cvtepi16\_storeu\_epi8(void \*, \_\_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.

## VPMULTISHIFTQB—Select Packed Unaligned Bytes From Quadword Sources

| Opcode / Instruction  | Op/En | 64/32 bit Mode Support | CPUID Feature Flag      | Description   |
|---|-------|------------------------|-------------------------|---|
| EVEX.128.66.0F38.W1 83 /r<br>VPMULTISHIFTQB xmm1 {k1}{z},<br>xmm2,xmm3/m128/m64bcst | A     | V/V                    | AVX512_VBMI<br>AVX512VL | Select unaligned bytes from qwords in xmm3/m128/m64bcst using control bytes in xmm2, write byte results to xmm1 under k1. |
| EVEX.256.66.0F38.W1 83 /r<br>VPMULTISHIFTQB ymm1 {k1}{z},<br>ymm2,ymm3/m256/m64bcst | A     | V/V                    | AVX512_VBMI<br>AVX512VL | Select unaligned bytes from qwords in ymm3/m256/m64bcst using control bytes in ymm2, write byte results to ymm1 under k1. |
| EVEX.512.66.0F38.W1 83 /r<br>VPMULTISHIFTQB zmm1 {k1}{z},<br>zmm2,zmm3/m512/m64bcst | A     | V/V                    | AVX512_VBMI             | Select unaligned bytes from qwords in zmm3/m512/m64bcst using control bytes in zmm2, write byte results to zmm1 under k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

#### 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[j].byte[j] &amp; 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;

ENDFOR

ENDFOR

DEST.qword[MAX\_VL-1:VL] := 0;

**Intel C/C++ Compiler Intrinsic Equivalent**

VPMULTISHIFTQB \_\_m512i \_\_mm512\_multishift\_epi64\_epi8( \_\_m512i a, \_\_m512i b);

VPMULTISHIFTQB \_\_m512i \_\_mm512\_mask\_multishift\_epi64\_epi8( \_\_m512i s, \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMULTISHIFTQB \_\_m512i \_\_mm512\_maskz\_multishift\_epi64\_epi8( \_\_mmask64 k, \_\_m512i a, \_\_m512i b);

VPMULTISHIFTQB \_\_m256i \_\_mm256\_multishift\_epi64\_epi8( \_\_m256i a, \_\_m256i b);

VPMULTISHIFTQB \_\_m256i \_\_mm256\_mask\_multishift\_epi64\_epi8( \_\_m256i s, \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMULTISHIFTQB \_\_m256i \_\_mm256\_maskz\_multishift\_epi64\_epi8( \_\_mmask32 k, \_\_m256i a, \_\_m256i b);

VPMULTISHIFTQB \_\_m128i \_\_mm\_multishift\_epi64\_epi8( \_\_m128i a, \_\_m128i b);

VPMULTISHIFTQB \_\_m128i \_\_mm\_mask\_multishift\_epi64\_epi8( \_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

VPMULTISHIFTQB \_\_m128i \_\_mm\_maskz\_multishift\_epi64\_epi8( \_\_mmask8 k, \_\_m128i a, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-50, "Type E4NF Class Exception Conditions."

**VPOPCNT—Return the Count of Number of Bits Set to 1 in BYTE/WORD/DWORD/QWORD**

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature Flag           | Description  |
|---|-----------|------------------------------|------------------------------|--|
| EVEX.128.66.0F38.W0 54 /r<br>VPOPCNTB xmm1{k1}{z},<br>xmm2/m128         | A         | V/V                          | AVX512_BITALG<br>AVX512VL    | Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.         |
| EVEX.256.66.0F38.W0 54 /r<br>VPOPCNTB ymm1{k1}{z},<br>ymm2/m256         | A         | V/V                          | AVX512_BITALG<br>AVX512VL    | Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.         |
| EVEX.512.66.0F38.W0 54 /r<br>VPOPCNTB zmm1{k1}{z},<br>zmm2/m512         | A         | V/V                          | AVX512_BITALG                | Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.         |
| EVEX.128.66.0F38.W1 54 /r<br>VPOPCNTW xmm1{k1}{z},<br>xmm2/m128         | A         | V/V                          | AVX512_BITALG<br>AVX512VL    | Counts the number of bits set to one in xmm2/m128 and puts the result in xmm1 with writemask k1.         |
| EVEX.256.66.0F38.W1 54 /r<br>VPOPCNTW ymm1{k1}{z},<br>ymm2/m256         | A         | V/V                          | AVX512_BITALG<br>AVX512VL    | Counts the number of bits set to one in ymm2/m256 and puts the result in ymm1 with writemask k1.         |
| EVEX.512.66.0F38.W1 54 /r<br>VPOPCNTW zmm1{k1}{z},<br>zmm2/m512         | A         | V/V                          | AVX512_BITALG                | Counts the number of bits set to one in zmm2/m512 and puts the result in zmm1 with writemask k1.         |
| EVEX.128.66.0F38.W0 55 /r<br>VPOPCNTD xmm1{k1}{z},<br>xmm2/m128/m32bcst | B         | V/V                          | AVX512_VPOPCNTDQ<br>AVX512VL | Counts the number of bits set to one in xmm2/m128/m32bcst and puts the result in xmm1 with writemask k1. |
| EVEX.256.66.0F38.W0 55 /r<br>VPOPCNTD ymm1{k1}{z},<br>ymm2/m256/m32bcst | B         | V/V                          | AVX512_VPOPCNTDQ<br>AVX512VL | Counts the number of bits set to one in ymm2/m256/m32bcst and puts the result in ymm1 with writemask k1. |
| EVEX.512.66.0F38.W0 55 /r<br>VPOPCNTD zmm1{k1}{z},<br>zmm2/m512/m32bcst | B         | V/V                          | AVX512_VPOPCNTDQ             | Counts the number of bits set to one in zmm2/m512/m32bcst and puts the result in zmm1 with writemask k1. |
| EVEX.128.66.0F38.W1 55 /r<br>VPOPCNTQ xmm1{k1}{z},<br>xmm2/m128/m64bcst | B         | V/V                          | AVX512_VPOPCNTDQ<br>AVX512VL | Counts the number of bits set to one in xmm2/m128/m64bcst and puts the result in xmm1 with writemask k1. |
| EVEX.256.66.0F38.W1 55 /r<br>VPOPCNTQ ymm1{k1}{z},<br>ymm2/m256/m64bcst | B         | V/V                          | AVX512_VPOPCNTDQ<br>AVX512VL | Counts the number of bits set to one in ymm2/m256/m64bcst and puts the result in ymm1 with writemask k1. |
| EVEX.512.66.0F38.W1 55 /r<br>VPOPCNTQ zmm1{k1}{z},<br>zmm2/m512/m64bcst | B         | V/V                          | AVX512_VPOPCNTDQ             | Counts the number of bits set to one in zmm2/m512/m64bcst and puts the result in zmm1 with writemask k1. |

**Instruction Operand Encoding**

| Op/En | Tuple    | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|----------|---------------|---------------|-----------|-----------|
| A     | Full Mem | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |
| B     | Full     | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

**Description**

This instruction counts the number of bits set to one in each byte, word, dword or qword element of its source (e.g., zmm2 or memory) and places the results in the destination register (zmm1). This instruction supports memory fault suppression.

**Operation****VPOPCNTB**

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1:

```

  IF MaskBit(j) OR *no writemask*:
    DEST.byte[j] := POPCNT(SRC.byte[j])
  ELSE IF *merging-masking*:
    *DEST.byte[j] remains unchanged*
  ELSE:
    DEST.byte[j] := 0
DEST[MAX_VL-1:VL] := 0

```

**VPOPCNTW**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:

```

  IF MaskBit(j) OR *no writemask*:
    DEST.word[j] := POPCNT(SRC.word[j])
  ELSE IF *merging-masking*:
    *DEST.word[j] remains unchanged*
  ELSE:
    DEST.word[j] := 0
DEST[MAX_VL-1:VL] := 0

```

**VPOPCNTD**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1:

```

  IF MaskBit(j) OR *no writemask*:
    IF SRC is broadcast memop:
      t := SRC.dword[0]
    ELSE:
      t := SRC.dword[j]
    DEST.dword[j] := POPCNT(t)
  ELSE IF *merging-masking*:
    *DEST.dword[j] remains unchanged*
  ELSE:
    DEST.dword[j] := 0
DEST[MAX_VL-1:VL] := 0

```

**VPOPCNTQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1:

```

  IF MaskBit(j) OR *no writemask*:
    IF SRC is broadcast memop:
      t := SRC.qword[0]
    ELSE:
      t := SRC.qword[j]
    DEST.qword[j] := POPCNT(t)
  ELSE IF *merging-masking*:
    *DEST.qword[j] remains unchanged*
  ELSE:
    DEST.qword[j] := 0
DEST[MAX_VL-1:VL] := 0

```



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPOPCNTW __m128i _mm_popcnt_epi16(__m128i);
VPOPCNTW __m128i _mm_mask_popcnt_epi16(__m128i, __mmask8, __m128i);
VPOPCNTW __m128i _mm_maskz_popcnt_epi16(__mmask8, __m128i);
VPOPCNTW __m256i _mm256_popcnt_epi16(__m256i);
VPOPCNTW __m256i _mm256_mask_popcnt_epi16(__m256i, __mmask16, __m256i);
VPOPCNTW __m256i _mm256_maskz_popcnt_epi16(__mmask16, __m256i);
VPOPCNTW __m512i _mm512_popcnt_epi16(__m512i);
VPOPCNTW __m512i _mm512_mask_popcnt_epi16(__m512i, __mmask32, __m512i);
VPOPCNTW __m512i _mm512_maskz_popcnt_epi16(__mmask32, __m512i);
VPOPCNTQ __m128i _mm_popcnt_epi64(__m128i);
VPOPCNTQ __m128i _mm_mask_popcnt_epi64(__m128i, __mmask8, __m128i);
VPOPCNTQ __m128i _mm_maskz_popcnt_epi64(__mmask8, __m128i);
VPOPCNTQ __m256i _mm256_popcnt_epi64(__m256i);
VPOPCNTQ __m256i _mm256_mask_popcnt_epi64(__m256i, __mmask8, __m256i);
VPOPCNTQ __m256i _mm256_maskz_popcnt_epi64(__mmask8, __m256i);
VPOPCNTQ __m512i _mm512_popcnt_epi64(__m512i);
VPOPCNTQ __m512i _mm512_mask_popcnt_epi64(__m512i, __mmask8, __m512i);
VPOPCNTQ __m512i _mm512_maskz_popcnt_epi64(__mmask8, __m512i);
VPOPCNTD __m128i _mm_popcnt_epi32(__m128i);
VPOPCNTD __m128i _mm_mask_popcnt_epi32(__m128i, __mmask8, __m128i);
VPOPCNTD __m128i _mm_maskz_popcnt_epi32(__mmask8, __m128i);
VPOPCNTD __m256i _mm256_popcnt_epi32(__m256i);
VPOPCNTD __m256i _mm256_mask_popcnt_epi32(__m256i, __mmask8, __m256i);
VPOPCNTD __m256i _mm256_maskz_popcnt_epi32(__mmask8, __m256i);
VPOPCNTD __m512i _mm512_popcnt_epi32(__m512i);
VPOPCNTD __m512i _mm512_mask_popcnt_epi32(__m512i, __mmask16, __m512i);
VPOPCNTD __m512i _mm512_maskz_popcnt_epi32(__mmask16, __m512i);
VPOPCNTB __m128i _mm_popcnt_epi8(__m128i);
VPOPCNTB __m128i _mm_mask_popcnt_epi8(__m128i, __mmask16, __m128i);
VPOPCNTB __m128i _mm_maskz_popcnt_epi8(__mmask16, __m128i);
VPOPCNTB __m256i _mm256_popcnt_epi8(__m256i);
VPOPCNTB __m256i _mm256_mask_popcnt_epi8(__m256i, __mmask32, __m256i);
VPOPCNTB __m256i _mm256_maskz_popcnt_epi8(__mmask32, __m256i);
VPOPCNTB __m512i _mm512_popcnt_epi8(__m512i);
VPOPCNTB __m512i _mm512_mask_popcnt_epi8(__m512i, __mmask64, __m512i);
VPOPCNTB __m512i _mm512_maskz_popcnt_epi8(__mmask64, __m512i);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, “Type E4 Class Exception Conditions.”

## VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left

| Opcode/<br>Instruction  | Op / En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|-----------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 15 /r<br>VPROLVD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | B       | V/V                         | AVX512VL<br>AVX512F      | Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.    |
| EVEX.128.66.0F.W0 72 /1 ib<br>VPROLD xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8 | A       | V/V                         | AVX512VL<br>AVX512F      | Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.  |
| EVEX.128.66.0F38.W1 15 /r<br>VPROLVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | B       | V/V                         | AVX512VL<br>AVX512F      | Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.      |
| EVEX.128.66.0F.W1 72 /1 ib<br>VPROLQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8 | A       | V/V                         | AVX512VL<br>AVX512F      | Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.  |
| EVEX.256.66.0F38.W0 15 /r<br>VPROLVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | B       | V/V                         | AVX512VL<br>AVX512F      | Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.    |
| EVEX.256.66.0F.W0 72 /1 ib<br>VPROLD ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8 | A       | V/V                         | AVX512VL<br>AVX512F      | Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.  |
| EVEX.256.66.0F38.W1 15 /r<br>VPROLVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | B       | V/V                         | AVX512VL<br>AVX512F      | Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.      |
| EVEX.256.66.0F.W1 72 /1 ib<br>VPROLQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8 | A       | V/V                         | AVX512VL<br>AVX512F      | Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.  |
| EVEX.512.66.0F38.W0 15 /r<br>VPROLVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | B       | V/V                         | AVX512F                  | Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1. |
| EVEX.512.66.0F.W0 72 /1 ib<br>VPROLD zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8 | A       | V/V                         | AVX512F                  | Rotate left of doublewords in zmm2/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.                                       |
| EVEX.512.66.0F38.W1 15 /r<br>VPROLVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | B       | V/V                         | AVX512F                  | Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1 under writemask k1.      |
| EVEX.512.66.0F.W1 72 /1 ib<br>VPROLQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8 | A       | V/V                         | AVX512F                  | Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.  |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full       | VEX.vvvv (w)  | ModRM:r/m (R) | imm8          | N/A       |
| B     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## 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;
  FI;
```

```

ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
        DEST[j+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[i+63:i])
        FI;
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

### Intel C/C++ Compiler Intrinsic Equivalent

```

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);
VPROLQ __m512i _mm512_rol_epi64(__m512i a, int imm);
VPROLQ __m512i _mm512_mask_rol_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPROLQ __m512i _mm512_maskz_rol_epi64(__mmask8 k, __m512i a, int imm);
VPROLQ __m256i _mm256_rol_epi64(__m256i a, int imm);
VPROLQ __m256i _mm256_mask_rol_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLQ __m256i _mm256_maskz_rol_epi64(__mmask8 k, __m256i a, int imm);
VPROLQ __m128i _mm_rol_epi64(__m128i a, int imm);
VPROLQ __m128i _mm_mask_rol_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLQ __m128i _mm_maskz_rol_epi64(__mmask8 k, __m128i a, int imm);
VPROLVD __m512i _mm512_rolv_epi32(__m512i a, __m512i cnt);
VPROLVD __m512i _mm512_mask_rolv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPROLVD __m512i _mm512_maskz_rolv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPROLVD __m256i _mm256_rolv_epi32(__m256i a, __m256i cnt);
VPROLVD __m256i _mm256_mask_rolv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLVD __m256i _mm256_maskz_rolv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPROLVD __m128i _mm_rolv_epi32(__m128i a, __m128i cnt);
VPROLVD __m128i _mm_mask_rolv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVD __m128i _mm_maskz_rolv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPROLVQ __m512i _mm512_rolv_epi64(__m512i a, __m512i cnt);
VPROLVQ __m512i _mm512_mask_rolv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPROLVQ __m512i _mm512_maskz_rolv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPROLVQ __m256i _mm256_rolv_epi64(__m256i a, __m256i cnt);
VPROLVQ __m256i _mm256_mask_rolv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLVQ __m256i _mm256_maskz_rolv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPROLVQ __m128i _mm_rolv_epi64(__m128i a, __m128i 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

| Opcode/<br>Instruction  | Op / En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 14 /r<br>VPRORVD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | B       | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1. |
| EVEX.128.66.0F.W0 72 /0 ib<br>VPRORD xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8 | A       | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.                                       |
| EVEX.128.66.0F38.W1 14 /r<br>VPRORVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | B       | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.   |
| EVEX.128.66.0F.W1 72 /0 ib<br>VPRORQ xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8 | A       | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.   |
| EVEX.256.66.0F38.W0 14 /r<br>VPRORVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | B       | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store using result writemask k1. |
| EVEX.256.66.0F.W0 72 /0 ib<br>VPRORD ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8 | A       | V/V                          | AVX512VL<br>AVX512F      | Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.                                       |
| EVEX.256.66.0F38.W1 14 /r<br>VPRORVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | B       | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.   |
| EVEX.256.66.0F.W1 72 /0 ib<br>VPRORQ ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8 | A       | V/V                          | AVX512VL<br>AVX512F      | Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.   |
| EVEX.512.66.0F38.W0 14 /r<br>VPRORVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | B       | V/V                          | AVX512F                  | Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1. |
| EVEX.512.66.0F.W0 72 /0 ib<br>VPRORD zmm1 {k1}{z},<br>zmm2/m512/m32bcst, imm8 | A       | V/V                          | AVX512F                  | Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.                                       |
| EVEX.512.66.0F38.W1 14 /r<br>VPRORVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | B       | V/V                          | AVX512F                  | Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.   |
| EVEX.512.66.0F.W1 72 /0 ib<br>VPRORQ zmm1 {k1}{z},<br>zmm2/m512/m64bcst, imm8 | A       | V/V                          | AVX512F                  | Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.   |

## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full       | VEX.vvvv (w)  | ModRM:r/m (R) | imm8          | N/A       |
| B     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## 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

```
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 32;
DEST[31:0] := (SRC >> COUNT) | (SRC << (32 - COUNT));
```

```
RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT := COUNT_SRC modulo 64;
DEST[63:0] := (SRC >> COUNT) | (SRC << (64 - COUNT));
```

### VPRORD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
      THEN DEST[i+31:i] := RIGHT_ROTATE_DWORDS( SRC1[31:0], imm8)
      ELSE DEST[i+31:i] := RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0
```

### VPRORVD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
  i := j * 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;
  FI;
```

```

ELSE
    IF *merging-masking*           ; merging-masking
        THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
        DEST[j+31:i] := 0
    FI
FI;
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*           ; 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

```

**VPRORVQ (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

```

    i := j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
            ELSE DEST[i+63:i] := RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
        FI;
    ELSE
        IF *merging-masking*           ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking*         ; zeroing-masking
            DEST[i+63:i] := 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPRORD __m512i _mm512_ror_epi32(__m512i a, int imm);
VPRORD __m512i _mm512_mask_ror_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPRORD __m512i _mm512_maskz_ror_epi32(__mmask16 k, __m512i a, int imm);
VPRORD __m256i _mm256_ror_epi32(__m256i a, int imm);
VPRORD __m256i _mm256_mask_ror_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORD __m256i _mm256_maskz_ror_epi32(__mmask8 k, __m256i a, int imm);
VPRORD __m128i _mm_ror_epi32(__m128i a, int imm);
VPRORD __m128i _mm_mask_ror_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORD __m128i _mm_maskz_ror_epi32(__mmask8 k, __m128i a, int imm);
VPRORQ __m512i _mm512_ror_epi64(__m512i a, int imm);
VPRORQ __m512i _mm512_mask_ror_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPRORQ __m512i _mm512_maskz_ror_epi64(__mmask8 k, __m512i a, int imm);
VPRORQ __m256i _mm256_ror_epi64(__m256i a, int imm);
VPRORQ __m256i _mm256_mask_ror_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORQ __m256i _mm256_maskz_ror_epi64(__mmask8 k, __m256i a, int imm);
VPRORQ __m128i _mm_ror_epi64(__m128i a, int imm);
VPRORQ __m128i _mm_mask_ror_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORQ __m128i _mm_maskz_ror_epi64(__mmask8 k, __m128i a, int imm);
VPRORVD __m512i _mm512_rorv_epi32(__m512i a, __m512i cnt);
VPRORVD __m512i _mm512_mask_rorv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPRORVD __m512i _mm512_maskz_rorv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPRORVD __m256i _mm256_rorv_epi32(__m256i a, __m256i cnt);
VPRORVD __m256i _mm256_mask_rorv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVD __m256i _mm256_maskz_rorv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPRORVD __m128i _mm_rorv_epi32(__m128i a, __m128i cnt);
VPRORVD __m128i _mm_mask_rorv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVD __m128i _mm_maskz_rorv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPRORVQ __m512i _mm512_rorv_epi64(__m512i a, __m512i cnt);
VPRORVQ __m512i _mm512_mask_rorv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORVQ __m512i _mm512_maskz_rorv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPRORVQ __m256i _mm256_rorv_epi64(__m256i a, __m256i cnt);
VPRORVQ __m256i _mm256_mask_rorv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVQ __m256i _mm256_maskz_rorv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPRORVQ __m128i _mm_rorv_epi64(__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

| Opcode/<br>Instruction                                       | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|-----------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 A0 /vsib<br>VPSCATTERDD vm32x {k1}, xmm1 | A         | V/V                         | AVX512VL<br>AVX512F      | Using signed dword indices, scatter dword values to memory using writemask k1. |
| EVEX.256.66.0F38.W0 A0 /vsib<br>VPSCATTERDD vm32y {k1}, ymm1 | A         | V/V                         | AVX512VL<br>AVX512F      | Using signed dword indices, scatter dword values to memory using writemask k1. |
| EVEX.512.66.0F38.W0 A0 /vsib<br>VPSCATTERDD vm32z {k1}, zmm1 | A         | V/V                         | AVX512F                  | Using signed dword indices, scatter dword values to memory using writemask k1. |
| EVEX.128.66.0F38.W1 A0 /vsib<br>VPSCATTERDQ vm32x {k1}, xmm1 | A         | V/V                         | AVX512VL<br>AVX512F      | Using signed dword indices, scatter qword values to memory using writemask k1. |
| EVEX.256.66.0F38.W1 A0 /vsib<br>VPSCATTERDQ vm32x {k1}, ymm1 | A         | V/V                         | AVX512VL<br>AVX512F      | Using signed dword indices, scatter qword values to memory using writemask k1. |
| EVEX.512.66.0F38.W1 A0 /vsib<br>VPSCATTERDQ vm32y {k1}, zmm1 | A         | V/V                         | AVX512F                  | Using signed dword indices, scatter qword values to memory using writemask k1. |
| EVEX.128.66.0F38.W0 A1 /vsib<br>VPSCATTERQD vm64x {k1}, xmm1 | A         | V/V                         | AVX512VL<br>AVX512F      | Using signed qword indices, scatter dword values to memory using writemask k1. |
| EVEX.256.66.0F38.W0 A1 /vsib<br>VPSCATTERQD vm64y {k1}, xmm1 | A         | V/V                         | AVX512VL<br>AVX512F      | Using signed qword indices, scatter dword values to memory using writemask k1. |
| EVEX.512.66.0F38.W0 A1 /vsib<br>VPSCATTERQD vm64z {k1}, ymm1 | A         | V/V                         | AVX512F                  | Using signed qword indices, scatter dword values to memory using writemask k1. |
| EVEX.128.66.0F38.W1 A1 /vsib<br>VPSCATTERQQ vm64x {k1}, xmm1 | A         | V/V                         | AVX512VL<br>AVX512F      | Using signed qword indices, scatter qword values to memory using writemask k1. |
| EVEX.256.66.0F38.W1 A1 /vsib<br>VPSCATTERQQ vm64y {k1}, ymm1 | A         | V/V                         | AVX512VL<br>AVX512F      | Using signed qword indices, scatter qword values to memory using writemask k1. |
| EVEX.512.66.0F38.W1 A1 /vsib<br>VPSCATTERQQ vm64z {k1}, zmm1 | A         | V/V                         | AVX512F                  | Using signed qword indices, scatter qword values to memory using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1   | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---|---------------|-----------|-----------|
| A     | Tuple1 Scalar | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | ModRM:reg (r) | N/A       | N/A       |

### Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address `BASE_ADDR` and index vector `VINDEX`, with scale `SCALE`. The elements are specified via the `VSIB` (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also includes 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  $\text{disp8} * N$  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  $\text{EVEX.Z} = 1$ .

### 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  $\text{MEM}[\text{BASE\_ADDR} + \text{SignExtend}(\text{VINDEX}[i+31:i]) * \text{SCALE} + \text{DISP}] := \text{SRC}[i+31:i]$

    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  $\text{MEM}[\text{BASE\_ADDR} + \text{SignExtend}(\text{VINDEX}[k+31:k]) * \text{SCALE} + \text{DISP}] := \text{SRC}[i+63:i]$

    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 + (VINDEK[k+63:k]) * SCALE + DISP] := SRC[j+31:i]
    k1[j] := 0

  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0

```

```

VPSCATTERQQ (EVEX encoded versions)
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEK[j+63:j]) * SCALE + DISP] := SRC[j+63:i]

  FI;
ENDFOR
k1[MAX_KL-1:KL] := 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VPSCATTERDD void __mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void __mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void __mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void __mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void __mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void __mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void __mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void __mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void __mm_i32scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void __mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void __mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void __mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void __mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void __mm256_i64scatter_epi32(void * base, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void __mm_i64scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void __mm512_mask_i64scatter_epi32(void * base, __mmask8 k, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void __mm256_mask_i64scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void __mm_mask_i64scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void __mm512_i64scatter_epi64(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void __mm256_i64scatter_epi64(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void __mm_i64scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void __mm512_mask_i64scatter_epi64(void * base, __mmask8 k, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void __mm256_mask_i64scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void __mm_mask_i64scatter_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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag    | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W1 70 /r /ib<br>VPSHLDW xmm1{k1}{z}, xmm2,<br>xmm3/m128, imm8          | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1. |
| EVEX.256.66.0F3A.W1 70 /r /ib<br>VPSHLDW ymm1{k1}{z}, ymm2,<br>ymm3/m256, imm8          | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1. |
| EVEX.512.66.0F3A.W1 70 /r /ib<br>VPSHLDW zmm1{k1}{z}, zmm2,<br>zmm3/m512, imm8          | A         | V/V                          | AVX512_VBMI2             | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1. |
| EVEX.128.66.0F3A.W0 71 /r /ib<br>VPSHLDD xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8  | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1. |
| EVEX.256.66.0F3A.W0 71 /r /ib<br>VPSHLDD ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8  | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1. |
| EVEX.512.66.0F3A.W0 71 /r /ib<br>VPSHLDD zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8  | B         | V/V                          | AVX512_VBMI2             | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1. |
| EVEX.128.66.0F3A.W1 71 /r /ib<br>VPSHLDDQ xmm1{k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into xmm1. |
| EVEX.256.66.0F3A.W1 71 /r /ib<br>VPSHLDDQ ymm1{k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into ymm1. |
| EVEX.512.66.0F3A.W1 71 /r /ib<br>VPSHLDDQ zmm1{k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | B         | V/V                          | AVX512_VBMI2             | Concatenate destination and source operands, extract result shifted to the left by constant value in imm8 into zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|----------|---------------|---------------|---------------|-----------|
| A     | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8 (r)  |
| B     | Full     | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8 (r)  |

### Description

Concatenate packed data, extract result shifted to the left by constant value.

This instruction supports memory fault suppression.

**Operation****VPSHLDW DEST, SRC2, SRC3, imm8**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:

IF MaskBit(j) OR \*no writemask\*:

tmp := concat(SRC2.word[j], SRC3.word[j]) &lt;&lt; (imm8 &amp; 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) &lt;&lt; (imm8 &amp; 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) &lt;&lt; (imm8 &amp; 63)

DEST.qword[j] := tmp.qword[1]

ELSE IF \*zeroing\*:

DEST.qword[j] := 0

\*ELSE DEST.qword[j] remains unchanged\*

DEST[MAX\_VL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSHLDD __m128i _mm_shldi_epi32(__m128i, __m128i, int);
VPSHLDD __m128i _mm_mask_shldi_epi32(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDD __m128i _mm_maskz_shldi_epi32(__mmask8, __m128i, __m128i, int);
VPSHLDD __m256i _mm256_shldi_epi32(__m256i, __m256i, int);
VPSHLDD __m256i _mm256_mask_shldi_epi32(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDD __m256i _mm256_maskz_shldi_epi32(__mmask8, __m256i, __m256i, int);
VPSHLDD __m512i _mm512_shldi_epi32(__m512i, __m512i, int);
VPSHLDD __m512i _mm512_mask_shldi_epi32(__m512i, __mmask16, __m512i, __m512i, int);
VPSHLDD __m512i _mm512_maskz_shldi_epi32(__mmask16, __m512i, __m512i, int);
VPSHLDDQ __m128i _mm_shldi_epi64(__m128i, __m128i, int);
VPSHLDDQ __m128i _mm_mask_shldi_epi64(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDDQ __m128i _mm_maskz_shldi_epi64(__mmask8, __m128i, __m128i, int);
VPSHLDDQ __m256i _mm256_shldi_epi64(__m256i, __m256i, int);
VPSHLDDQ __m256i _mm256_mask_shldi_epi64(__m256i, __mmask8, __m256i, __m256i, int);
VPSHLDDQ __m256i _mm256_maskz_shldi_epi64(__mmask8, __m256i, __m256i, int);
VPSHLDDQ __m512i _mm512_shldi_epi64(__m512i, __m512i, int);
VPSHLDDQ __m512i _mm512_mask_shldi_epi64(__m512i, __mmask8, __m512i, __m512i, int);
VPSHLDDQ __m512i _mm512_maskz_shldi_epi64(__mmask8, __m512i, __m512i, int);
VPSHLDDW __m128i _mm_shldi_epi16(__m128i, __m128i, int);
VPSHLDDW __m128i _mm_mask_shldi_epi16(__m128i, __mmask8, __m128i, __m128i, int);
VPSHLDDW __m128i _mm_maskz_shldi_epi16(__mmask8, __m128i, __m128i, int);
VPSHLDDW __m256i _mm256_shldi_epi16(__m256i, __m256i, int);
VPSHLDDW __m256i _mm256_mask_shldi_epi16(__m256i, __mmask16, __m256i, __m256i, int);
VPSHLDDW __m256i _mm256_maskz_shldi_epi16(__mmask16, __m256i, __m256i, int);
VPSHLDDW __m512i _mm512_shldi_epi16(__m512i, __m512i, int);
VPSHLDDW __m512i _mm512_mask_shldi_epi16(__m512i, __mmask32, __m512i, __m512i, int);
VPSHLDDW __m512i _mm512_maskz_shldi_epi16(__mmask32, __m512i, __m512i, int);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, “Type E4 Class Exception Conditions.”



## VPSHLDV—Concatenate and Variable Shift Packed Data Left Logical

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag    | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W1 70 /r<br>VPSHLDVW xmm1{k1}{z}, xmm2,<br>xmm3/m128         | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1. |
| EVEX.256.66.0F38.W1 70 /r<br>VPSHLDVW ymm1{k1}{z}, ymm2,<br>ymm3/m256         | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1. |
| EVEX.512.66.0F38.W1 70 /r<br>VPSHLDVW zmm1{k1}{z}, zmm2,<br>zmm3/m512         | A         | V/V                          | AVX512_VBMI2             | Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1. |
| EVEX.128.66.0F38.W0 71 /r<br>VPSHLDVD xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1. |
| EVEX.256.66.0F38.W0 71 /r<br>VPSHLDVD ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1. |
| EVEX.512.66.0F38.W0 71 /r<br>VPSHLDVD zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst | B         | V/V                          | AVX512_VBMI2             | Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1. |
| EVEX.128.66.0F38.W1 71 /r<br>VPSHLDVQ xmm1{k1}{z}, xmm2,<br>xmm3/m128/m64bcst | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the left by value in xmm3/m128 into xmm1. |
| EVEX.256.66.0F38.W1 71 /r<br>VPSHLDVQ ymm1{k1}{z}, ymm2,<br>ymm3/m256/m64bcst | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the left by value in xmm3/m256 into ymm1. |
| EVEX.512.66.0F38.W1 71 /r<br>VPSHLDVQ zmm1{k1}{z}, zmm2,<br>zmm3/m512/m64bcst | B         | V/V                          | AVX512_VBMI2             | Concatenate zmm1 and zmm2, extract result shifted to the left by value in zmm3/m512 into zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|----------|------------------|---------------|---------------|-----------|
| A     | Full Mem | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| B     | Full     | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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]) &lt;&lt; (SRC3.word[j] &amp; 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]) &lt;&lt; (tsrc3 &amp; 31)

DEST.dword[j] := tmp.dword[1]

ELSE IF \*zeroing\*:

DEST.dword[j] := 0

\*ELSE DEST.dword[j] remains unchanged\*

DEST[MAX\_VL-1:VL] := 0

**VPSHLDVQ DEST, SRC2, SRC3**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1:

IF SRC3 is broadcast memop:

tsrc3 := SRC3.qword[0]

ELSE:

tsrc3 := SRC3.qword[j]

IF MaskBit(j) OR \*no writemask\*:

tmp := concat(DEST.qword[j], SRC2.qword[j]) &lt;&lt; (tsrc3 &amp; 63)

DEST.qword[j] := tmp.qword[1]

ELSE IF \*zeroing\*:

DEST.qword[j] := 0

\*ELSE DEST.qword[j] remains unchanged\*

DEST[MAX\_VL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVQ __m512i _mm512_shldv_epi64(__m512i, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_mask_shldv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHLDVQ __m512i _mm512_maskz_shldv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHLDVW __m128i _mm_shldv_epi16(__m128i, __m128i, __m128i);
VPSHLDVW __m128i _mm_mask_shldv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVW __m128i _mm_maskz_shldv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVW __m256i _mm256_shldv_epi16(__m256i, __m256i, __m256i);
VPSHLDVW __m256i _mm256_mask_shldv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHLDVW __m256i _mm256_maskz_shldv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHLDVW __m512i _mm512_shldv_epi16(__m512i, __m512i, __m512i);
VPSHLDVW __m512i _mm512_mask_shldv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHLDVW __m512i _mm512_maskz_shldv_epi16(__mmask32, __m512i, __m512i, __m512i);
VPSHLDVD __m128i _mm_shldv_epi32(__m128i, __m128i, __m128i);
VPSHLDVD __m128i _mm_mask_shldv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHLDVD __m128i _mm_maskz_shldv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHLDVD __m256i _mm256_shldv_epi32(__m256i, __m256i, __m256i);
VPSHLDVD __m256i _mm256_mask_shldv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHLDVD __m256i _mm256_maskz_shldv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHLDVD __m512i _mm512_shldv_epi32(__m512i, __m512i, __m512i);
VPSHLDVD __m512i _mm512_mask_shldv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHLDVD __m512i _mm512_maskz_shldv_epi32(__mmask16, __m512i, __m512i, __m512i);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions."

## VPSHRD—Concatenate and Shift Packed Data Right Logical

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag    | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W1 72 /r /ib<br>VPSHRDW xmm1{k1}{z}, xmm2,<br>xmm3/m128, imm8         | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1. |
| EVEX.256.66.0F3A.W1 72 /r /ib<br>VPSHRDW ymm1{k1}{z}, ymm2,<br>ymm3/m256, imm8         | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1. |
| EVEX.512.66.0F3A.W1 72 /r /ib<br>VPSHRDW zmm1{k1}{z}, zmm2,<br>zmm3/m512, imm8         | A         | V/V                          | AVX512_VBMI2             | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1. |
| EVEX.128.66.0F3A.W0 73 /r /ib<br>VPSHRDD xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1. |
| EVEX.256.66.0F3A.W0 73 /r /ib<br>VPSHRDD ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1. |
| EVEX.512.66.0F3A.W0 73 /r /ib<br>VPSHRDD zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8 | B         | V/V                          | AVX512_VBMI2             | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1. |
| EVEX.128.66.0F3A.W1 73 /r /ib<br>VPSHRDQ xmm1{k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into xmm1. |
| EVEX.256.66.0F3A.W1 73 /r /ib<br>VPSHRDQ ymm1{k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8 | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into ymm1. |
| EVEX.512.66.0F3A.W1 73 /r /ib<br>VPSHRDQ zmm1{k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | B         | V/V                          | AVX512_VBMI2             | Concatenate destination and source operands, extract result shifted to the right by constant value in imm8 into zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|----------|---------------|---------------|---------------|-----------|
| A     | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8 (r)  |
| B     | Full     | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8 (r)  |

### Description

Concatenate packed data, extract result shifted to the right by constant value.

This instruction supports memory fault suppression.

**Operation****VPSHRDW DEST, SRC2, SRC3, imm8**

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1:

IF MaskBit(j) OR \*no writemask\*:

DEST.word[j] := concat(SRC3.word[j], SRC2.word[j]) &gt;&gt; (imm8 &amp; 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]) &gt;&gt; (imm8 &amp; 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]) &gt;&gt; (imm8 &amp; 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 Table 2-49, “Type E4 Class Exception Conditions.”

## VPSHRDV—Concatenate and Variable Shift Packed Data Right Logical

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag    | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 72 /r<br>VPSHRDVW xmm1{k1}{z}, xmm2,<br>xmm3/m128         | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1. |
| EVEX.256.66.0F38.W1 72 /r<br>VPSHRDVW ymm1{k1}{z}, ymm2,<br>ymm3/m256         | A         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1. |
| EVEX.512.66.0F38.W1 72 /r<br>VPSHRDVW zmm1{k1}{z}, zmm2,<br>zmm3/m512         | A         | V/V                          | AVX512_VBMI2             | Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1. |
| EVEX.128.66.0F38.W0 73 /r<br>VPSHRDVD xmm1{k1}{z}, xmm2,<br>xmm3/m128/m32bcst | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1. |
| EVEX.256.66.0F38.W0 73 /r<br>VPSHRDVD ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1. |
| EVEX.512.66.0F38.W0 73 /r<br>VPSHRDVD zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst | B         | V/V                          | AVX512_VBMI2             | Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1. |
| EVEX.128.66.0F38.W1 73 /r<br>VPSHRDVQ xmm1{k1}{z}, xmm2,<br>xmm3/m128/m64bcst | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate xmm1 and xmm2, extract result shifted to the right by value in xmm3/m128 into xmm1. |
| EVEX.256.66.0F38.W1 73 /r<br>VPSHRDVQ ymm1{k1}{z}, ymm2,<br>ymm3/m256/m64bcst | B         | V/V                          | AVX512_VBMI2<br>AVX512VL | Concatenate ymm1 and ymm2, extract result shifted to the right by value in xmm3/m256 into ymm1. |
| EVEX.512.66.0F38.W1 73 /r<br>VPSHRDVQ zmm1{k1}{z}, zmm2,<br>zmm3/m512/m64bcst | B         | V/V                          | AVX512_VBMI2             | Concatenate zmm1 and zmm2, extract result shifted to the right by value in zmm3/m512 into zmm1. |

### Instruction Operand Encoding

| Op/En | Tuple    | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|----------|------------------|---------------|---------------|-----------|
| A     | Full Mem | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| B     | Full     | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

#### 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]) &gt;&gt; (SRC3.word[j] &amp; 15)

ELSE IF \*zeroing\*:

DEST.word[j] := 0

\*ELSE DEST.word[j] remains unchanged\*

DEST[MAX\_VL-1:VL] := 0

**VPSHRDVD DEST, SRC2, SRC3**

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1:

IF SRC3 is broadcast memop:

tsrc3 := SRC3.dword[0]

ELSE:

tsrc3 := SRC3.dword[j]

IF MaskBit(j) OR \*no writemask\*:

DEST.dword[j] := concat(SRC2.dword[j], DEST.dword[j]) &gt;&gt; (tsrc3 &amp; 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]) &gt;&gt; (tsrc3 &amp; 63)

ELSE IF \*zeroing\*:

DEST.qword[j] := 0

\*ELSE DEST.qword[j] remains unchanged\*

DEST[MAX\_VL-1:VL] := 0



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VPSHRDVQ __m128i __mm_shrdv_epi64(__m128i, __m128i, __m128i);
VPSHRDVQ __m128i __mm_mask_shrdv_epi64(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVQ __m128i __mm_maskz_shrdv_epi64(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVQ __m256i __mm256_shrdv_epi64(__m256i, __m256i, __m256i);
VPSHRDVQ __m256i __mm256_mask_shrdv_epi64(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVQ __m256i __mm256_maskz_shrdv_epi64(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVQ __m512i __mm512_shrdv_epi64(__m512i, __m512i, __m512i);
VPSHRDVQ __m512i __mm512_mask_shrdv_epi64(__m512i, __mmask8, __m512i, __m512i);
VPSHRDVQ __m512i __mm512_maskz_shrdv_epi64(__mmask8, __m512i, __m512i, __m512i);
VPSHRDVD __m128i __mm_shrdv_epi32(__m128i, __m128i, __m128i);
VPSHRDVD __m128i __mm_mask_shrdv_epi32(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVD __m128i __mm_maskz_shrdv_epi32(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVD __m256i __mm256_shrdv_epi32(__m256i, __m256i, __m256i);
VPSHRDVD __m256i __mm256_mask_shrdv_epi32(__m256i, __mmask8, __m256i, __m256i);
VPSHRDVD __m256i __mm256_maskz_shrdv_epi32(__mmask8, __m256i, __m256i, __m256i);
VPSHRDVD __m512i __mm512_shrdv_epi32(__m512i, __m512i, __m512i);
VPSHRDVD __m512i __mm512_mask_shrdv_epi32(__m512i, __mmask16, __m512i, __m512i);
VPSHRDVD __m512i __mm512_maskz_shrdv_epi32(__mmask16, __m512i, __m512i, __m512i);
VPSHRDVW __m128i __mm_shrdv_epi16(__m128i, __m128i, __m128i);
VPSHRDVW __m128i __mm_mask_shrdv_epi16(__m128i, __mmask8, __m128i, __m128i);
VPSHRDVW __m128i __mm_maskz_shrdv_epi16(__mmask8, __m128i, __m128i, __m128i);
VPSHRDVW __m256i __mm256_shrdv_epi16(__m256i, __m256i, __m256i);
VPSHRDVW __m256i __mm256_mask_shrdv_epi16(__m256i, __mmask16, __m256i, __m256i);
VPSHRDVW __m256i __mm256_maskz_shrdv_epi16(__mmask16, __m256i, __m256i, __m256i);
VPSHRDVW __m512i __mm512_shrdv_epi16(__m512i, __m512i, __m512i);
VPSHRDVW __m512i __mm512_mask_shrdv_epi16(__m512i, __mmask32, __m512i, __m512i);
VPSHRDVW __m512i __mm512_maskz_shrdv_epi16(__mmask32, __m512i, __m512i, __m512i);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, “Type E4 Class Exception Conditions.”

## VPSHUFBITQMB—Shuffle Bits From Quadword Elements Using Byte Indexes Into Mask

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag     | Description  |
|--|-----------|------------------------------|---------------------------|--|
| EVEX.128.66.0F38.W0 8F /r<br>VPSHUFBITQMB k1{k2}, xmm2,<br>xmm3/m128 | A         | V/V                          | AVX512_BITALG<br>AVX512VL | Extract values in xmm2 using control bits of<br>xmm3/m128 with writemask k2 and leave the<br>result in mask register k1. |
| EVEX.256.66.0F38.W0 8F /r<br>VPSHUFBITQMB k1{k2}, ymm2,<br>ymm3/m256 | A         | V/V                          | AVX512_BITALG<br>AVX512VL | Extract values in ymm2 using control bits of<br>ymm3/m256 with writemask k2 and leave the<br>result in mask register k1. |
| EVEX.512.66.0F38.W0 8F /r<br>VPSHUFBITQMB k1{k2}, zmm2,<br>zmm3/m512 | A         | V/V                          | AVX512_BITALG             | Extract values in zmm2 using control bits of<br>zmm3/m512 with writemask k2 and leave the<br>result in mask register k1. |

### Instruction Operand Encoding

| Op/En | Tuple    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|----------|---------------|---------------|---------------|-----------|
| A     | Full Mem | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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[\*8+j] or \*no writemask\*:

m := SRC2.qword[i].byte[j] & 0x3F

k1[\*8+j] := SRC1.qword[i].bit[m]

ELSE:

k1[\*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 __m512_bitshuffle_epi64_mask(__m512i, __m512i);
```

```
VPSHUFBITQMB __mmask64 __m512_mask_bitshuffle_epi64_mask(__mmask64, __m512i, __m512i);
```

## VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 47 /r<br>VPSLLVD xmm1, xmm2, xmm3/m128                     | A          | V/V                          | AVX2                     | Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.                            |
| VEX.128.66.0F38.W1 47 /r<br>VPSLLVQ xmm1, xmm2, xmm3/m128                     | A          | V/V                          | AVX2                     | Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.                              |
| VEX.256.66.0F38.W0 47 /r<br>VPSLLVD ymm1, ymm2, ymm3/m256                     | A          | V/V                          | AVX2                     | Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.                            |
| VEX.256.66.0F38.W1 47 /r<br>VPSLLVQ ymm1, ymm2, ymm3/m256                     | A          | V/V                          | AVX2                     | Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.                              |
| EVEX.128.66.0F38.W1 12 /r<br>VPSLLVW xmm1 {k1}{z}, xmm2,<br>xmm3/m128         | B          | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.               |
| EVEX.256.66.0F38.W1 12 /r<br>VPSLLVW ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | B          | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.               |
| EVEX.512.66.0F38.W1 12 /r<br>VPSLLVW zmm1 {k1}{z}, zmm2,<br>zmm3/m512         | B          | V/V                          | AVX512BW                 | Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.               |
| EVEX.128.66.0F38.W0 47 /r<br>VPSLLVD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | C          | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1. |
| EVEX.256.66.0F38.W0 47 /r<br>VPSLLVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | C          | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1. |
| EVEX.512.66.0F38.W0 47 /r<br>VPSLLVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | C          | V/V                          | AVX512F                  | Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1. |
| EVEX.128.66.0F38.W1 47 /r<br>VPSLLVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | C          | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.   |
| EVEX.256.66.0F38.W1 47 /r<br>VPSLLVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | C          | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.   |
| EVEX.512.66.0F38.W1 47 /r<br>VPSLLVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | C          | V/V                          | AVX512F                  | Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.   |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full Mem   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| C     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

**Description**

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation****VPSLLVW (EVEX encoded version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

FOR j := 0 TO KL-1
  i := j * 16
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] := ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+15:i] := 0
      FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;

```

**VPSLLVD (VEX.128 version)**

```

COUNT_0 := SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] := 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] := ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
DEST[127:96] := 0;
DEST[MAXVL-1:128] := 0;

```

**VPSLLVD (VEX.256 version)**

```

COUNT_0 := SRC2[31 : 0];
(* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 := SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] := ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] := 0;
(* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] := ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] := 0;
DEST[MAXVL-1:256] := 0;

```

**VPSLLVD (EVEX encoded version)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
      ELSE DEST[i+31:i] := ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] := 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;

```

**VPSLLVQ (VEX.128 version)**

```

COUNT_0 := SRC2[63 : 0];
COUNT_1 := SRC2[127 : 64];
IF COUNT_0 < 64 THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] := 0;
IF COUNT_1 < 64 THEN
DEST[127:64] := ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] := 0;
DEST[MAXVL-1:128] := 0;

```

**VPSLLVQ (VEX.256 version)**

```

COUNT_0 := SRC2[63 : 0];
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] := ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] := 0;
(* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] := ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] := 0;
DEST[MAXVL-1:256] := 0;

```

**VPSLLVQ (EVEX encoded version)**

```

(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
      ELSE DEST[i+63:i] := ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0;

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VPSLLVW \_\_m512i \_mm512\_sllv\_epi16(\_\_m512i a, \_\_m512i cnt);  
 VPSLLVW \_\_m512i \_mm512\_mask\_sllv\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVW \_\_m512i \_mm512\_maskz\_sllv\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVW \_\_m256i \_mm256\_mask\_sllv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVW \_\_m256i \_mm256\_maskz\_sllv\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVW \_\_m128i \_mm\_mask\_sllv\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVW \_\_m128i \_mm\_maskz\_sllv\_epi16(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m512i \_mm512\_sllv\_epi32(\_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m512i \_mm512\_mask\_sllv\_epi32(\_\_m512i s, \_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m512i \_mm512\_maskz\_sllv\_epi32(\_\_mmask16 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m256i \_mm256\_mask\_sllv\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m256i \_mm256\_maskz\_sllv\_epi32(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m128i \_mm\_mask\_sllv\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m128i \_mm\_maskz\_sllv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVQ \_\_m512i \_mm512\_sllv\_epi64(\_\_m512i a, \_\_m512i cnt);  
 VPSLLVQ \_\_m512i \_mm512\_mask\_sllv\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVQ \_\_m512i \_mm512\_maskz\_sllv\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSLLVD \_\_m256i \_mm256\_mask\_sllv\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m256i \_mm256\_maskz\_sllv\_epi64(\_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSLLVD \_\_m128i \_mm\_mask\_sllv\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m128i \_mm\_maskz\_sllv\_epi64(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSLLVD \_\_m256i \_mm256\_sllv\_epi32(\_\_m256i m, \_\_m256i count)  
 VPSLLVQ \_\_m256i \_mm256\_sllv\_epi64(\_\_m256i m, \_\_m256i count)

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

VEX-encoded instructions, see Table 2-21, “Type 4 Class Exception Conditions.”

EVEX-encoded VPSLLVD/VPSLLVQ, see Table 2-49, “Type E4 Class Exception Conditions.”

EVEX-encoded VPSLLVW, see Exceptions Type E4.nb in Table 2-49, “Type E4 Class Exception Conditions.”

## VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 46 /r<br>VPSRAVD xmm1, xmm2, xmm3/m128                     | A         | V/V                          | AVX2                     | Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.                            |
| VEX.256.66.0F38.W0 46 /r<br>VPSRAVD ymm1, ymm2, ymm3/m256                     | A         | V/V                          | AVX2                     | Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.                            |
| EVEX.128.66.0F38.W1 11 /r<br>VPSRAVW xmm1 {k1}{z}, xmm2,<br>xmm3/m128         | B         | V/V                          | AVX512VL<br>AVX512BW     | Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.               |
| EVEX.256.66.0F38.W1 11 /r<br>VPSRAVW ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | B         | V/V                          | AVX512VL<br>AVX512BW     | Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.               |
| EVEX.512.66.0F38.W1 11 /r<br>VPSRAVW zmm1 {k1}{z}, zmm2,<br>zmm3/m512         | B         | V/V                          | AVX512BW                 | Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.               |
| EVEX.128.66.0F38.W0 46 /r<br>VPSRAVD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | C         | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1. |
| EVEX.256.66.0F38.W0 46 /r<br>VPSRAVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | C         | V/V                          | AVX512VL<br>AVX512F      | Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1. |
| EVEX.512.66.0F38.W0 46 /r<br>VPSRAVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | C         | V/V                          | AVX512F                  | Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1. |
| EVEX.128.66.0F38.W1 46 /r<br>VPSRAVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | C         | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.   |
| EVEX.256.66.0F38.W1 46 /r<br>VPSRAVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | C         | V/V                          | AVX512VL<br>AVX512F      | Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.   |
| EVEX.512.66.0F38.W1 46 /r<br>VPSRAVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | C         | V/V                          | AVX512F                  | Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.   |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full Mem   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| C     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |



## 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[j+3:i]
      IF COUNT < 16
        THEN DEST[i+15:i] := SignExtend(SRC1[j+15:i] >> COUNT)
        ELSE
          FOR k := 0 TO 15
            DEST[i+k] := SRC1[i+15]
          ENDFOR;
        FI
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+15:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+15:i] := 0
          FI
        FI;
    ENDFOR;
  DEST[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;
          FI
        ELSE
          COUNT := SRC2[j+4:i]
          IF COUNT < 32
            THEN DEST[i+31:i] := SignExtend(SRC1[i+31:i] >> COUNT)
            ELSE
              FOR k := 0 TO 31
                DEST[i+k] := SRC1[i+31]
              ENDFOR;
            FI
          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 &lt; 64

THEN DEST[i+63:i] := SignExtend(SRC1[i+63:i] &gt;&gt; COUNT)

ELSE

FOR k := 0 TO 63

DEST[i+k] := SRC1[i+63]

ENDFOR;

FI

ELSE

COUNT := SRC2[j+5:i]

IF COUNT &lt; 64

THEN DEST[i+63:i] := SignExtend(SRC1[i+63:i] &gt;&gt; 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 ; zeroing-masking

DEST[63:0] := 0

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 __m256i __mm256_srav_epi32(__m256i a, __m256i cnt);
VPSRAVD __m256i __mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i __mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m128i __mm_srav_epi32(__m128i a, __m128i cnt);
VPSRAVD __m128i __mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m128i __mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m512i __mm512_srav_epi64(__m512i a, __m512i cnt);
VPSRAVQ __m512i __mm512_mask_srav_epi64(__m512i s, __mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i __mm512_maskz_srav_epi64(__mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m256i __mm256_srav_epi64(__m256i a, __m256i cnt);
VPSRAVQ __m256i __mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m256i __mm256_maskz_srav_epi64(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m128i __mm_srav_epi64(__m128i a, __m128i cnt);
VPSRAVQ __m128i __mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m128i __mm_maskz_srav_epi64(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m512i __mm512_srav_epi16(__m512i a, __m512i cnt);
VPSRAVW __m512i __mm512_mask_srav_epi16(__m512i s, __mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m512i __mm512_maskz_srav_epi16(__mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m256i __mm256_srav_epi16(__m256i a, __m256i cnt);
VPSRAVW __m256i __mm256_mask_srav_epi16(__m256i s, __mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m256i __mm256_maskz_srav_epi16(__mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m128i __mm_srav_epi16(__m128i a, __m128i cnt);
VPSRAVW __m128i __mm_mask_srav_epi16(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m128i __mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m256i __mm256_srav_epi32 (__m256i m, __m256i count)

```

**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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------------|--------------------------|--|
| VEX.128.66.0F38.W0 45 /r<br>VPSRLVD xmm1, xmm2, xmm3/m128                     | A         | V/V                         | AVX2                     | Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.                            |
| VEX.128.66.0F38.W1 45 /r<br>VPSRLVQ xmm1, xmm2, xmm3/m128                     | A         | V/V                         | AVX2                     | Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.                              |
| VEX.256.66.0F38.W0 45 /r<br>VPSRLVD ymm1, ymm2, ymm3/m256                     | A         | V/V                         | AVX2                     | Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.                            |
| VEX.256.66.0F38.W1 45 /r<br>VPSRLVQ ymm1, ymm2, ymm3/m256                     | A         | V/V                         | AVX2                     | Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.                              |
| EVEX.128.66.0F38.W1 10 /r<br>VPSRLVW xmm1 {k1}{z}, xmm2,<br>xmm3/m128         | B         | V/V                         | AVX512VL<br>AVX512BW     | Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.               |
| EVEX.256.66.0F38.W1 10 /r<br>VPSRLVW ymm1 {k1}{z}, ymm2,<br>ymm3/m256         | B         | V/V                         | AVX512VL<br>AVX512BW     | Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.               |
| EVEX.512.66.0F38.W1 10 /r<br>VPSRLVW zmm1 {k1}{z}, zmm2,<br>zmm3/m512         | B         | V/V                         | AVX512BW                 | Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.               |
| EVEX.128.66.0F38.W0 45 /r<br>VPSRLVD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst | C         | V/V                         | AVX512VL<br>AVX512F      | Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1. |
| EVEX.256.66.0F38.W0 45 /r<br>VPSRLVD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst | C         | V/V                         | AVX512VL<br>AVX512F      | Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1. |
| EVEX.512.66.0F38.W0 45 /r<br>VPSRLVD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst | C         | V/V                         | AVX512F                  | Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1. |
| EVEX.128.66.0F38.W1 45 /r<br>VPSRLVQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst | C         | V/V                         | AVX512VL<br>AVX512F      | Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.   |
| EVEX.256.66.0F38.W1 45 /r<br>VPSRLVQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst | C         | V/V                         | AVX512VL<br>AVX512F      | Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.   |
| EVEX.512.66.0F38.W1 45 /r<br>VPSRLVQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst | C         | V/V                         | AVX512F                  | Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.   |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | N/A        | ModRM:reg (w) | VEX.vvvv (r)  | ModRM:r/m (r) | N/A       |
| B     | Full Mem   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| C     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

**Description**

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSRLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSRLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

**Operation****VPSRLVW (EVEX encoded version)**

(KL, VL) = (8, 128), (16, 256), (32, 512)

```

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;

```

**VPSRLVD (VEX.128 version)**

```

COUNT_0 := SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 := SRC2[127 : 96];
IF COUNT_0 < 32 THEN
  DEST[31:0] := ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
  DEST[31:0] := 0;
  (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
  DEST[127:96] := ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
  DEST[127:96] := 0;
DEST[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;

```



**Intel C/C++ Compiler Intrinsic Equivalent**

VPSRLVW \_\_m512i \_mm512\_srlv\_epi16(\_\_m512i a, \_\_m512i cnt);  
 VPSRLVW \_\_m512i \_mm512\_mask\_srlv\_epi16(\_\_m512i s, \_\_mmask32 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVW \_\_m512i \_mm512\_maskz\_srlv\_epi16(\_\_mmask32 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVW \_\_m256i \_mm256\_mask\_srlv\_epi16(\_\_m256i s, \_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVW \_\_m256i \_mm256\_maskz\_srlv\_epi16(\_\_mmask16 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVW \_\_m128i \_mm\_mask\_srlv\_epi16(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVW \_\_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);  
 VPSRLVD \_\_m128i \_mm\_mask\_srlv\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVD \_\_m128i \_mm\_maskz\_srlv\_epi32(\_\_mmask8 k, \_\_m128i a, \_\_m128i cnt);  
 VPSRLVQ \_\_m512i \_mm512\_srlv\_epi64(\_\_m512i a, \_\_m512i cnt);  
 VPSRLVQ \_\_m512i \_mm512\_mask\_srlv\_epi64(\_\_m512i s, \_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVQ \_\_m512i \_mm512\_maskz\_srlv\_epi64(\_\_mmask8 k, \_\_m512i a, \_\_m512i cnt);  
 VPSRLVQ \_\_m256i \_mm256\_mask\_srlv\_epi64(\_\_m256i s, \_\_mmask8 k, \_\_m256i a, \_\_m256i cnt);  
 VPSRLVQ \_\_m256i \_mm256\_maskz\_srlv\_epi64(\_\_mmask8 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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|-----------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W0 25 /r ib<br>VPTERNLOGD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8 | A         | V/V                         | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F3A.W0 25 /r ib<br>VPTERNLOGD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8 | A         | V/V                         | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F3A.W0 25 /r ib<br>VPTERNLOGD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8 | A         | V/V                         | AVX512F                  | 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. |
| EVEX.128.66.0F3A.W1 25 /r ib<br>VPTERNLOGQ xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8 | A         | V/V                         | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F3A.W1 25 /r ib<br>VPTERNLOGQ ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8 | A         | V/V                         | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F3A.W1 25 /r ib<br>VPTERNLOGQ zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | A         | V/V                         | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1        | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|------------------|---------------|---------------|-----------|
| A     | Full       | ModRM:reg (r, w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### 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 5-12 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

**Table 5-12. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values**

| VPTERNLOGD reg1, reg2, src3, 0xE2 |           |           | Bit Result with Imm8=0xE2 | VPTERNLOGD reg1, reg2, src3, 0xE4 |           |           | Bit Result with Imm8=0xE4 |
|-----------------------------------|-----------|-----------|---------------------------|-----------------------------------|-----------|-----------|---------------------------|
| Bit(reg1)                         | Bit(reg2) | Bit(src3) |                           | Bit(reg1)                         | Bit(reg2) | Bit(src3) |                           |
| 0                                 | 0         | 0         | 0                         | 0                                 | 0         | 0         | 0                         |
| 0                                 | 0         | 1         | 1                         | 0                                 | 0         | 1         | 0                         |
| 0                                 | 1         | 0         | 0                         | 0                                 | 1         | 0         | 1                         |
| 0                                 | 1         | 1         | 0                         | 0                                 | 1         | 1         | 0                         |
| 1                                 | 0         | 0         | 0                         | 1                                 | 0         | 0         | 0                         |
| 1                                 | 0         | 1         | 1                         | 1                                 | 0         | 1         | 1                         |
| 1                                 | 1         | 0         | 1                         | 1                                 | 1         | 0         | 1                         |
| 1                                 | 1         | 1         | 1                         | 1                                 | 1         | 1         | 1                         |

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-1 and Table 5-2 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

### Operation

#### VPTERNLOGD (EVEX encoded versions)

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

  i := j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

      FOR k := 0 TO 31

        IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

          THEN DEST[j][k] := imm[(DEST[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;

      ; 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] &lt;&lt; 2) + (SRC1[i+k] &lt;&lt; 1) + SRC2[k]]

ELSE DEST[j][k] := imm[(DEST[i+k] &lt;&lt; 2) + (SRC1[i+k] &lt;&lt; 1) + SRC2[i+k]]

FI; ; table lookup of immediate below;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63+i:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[63+i:i] := 0

FI;

FI;

ENDFOR;

DEST[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);

VPTERNLOGQ \_\_m512i \_\_mm512\_ternarylogic\_epi64(\_\_m512i a, \_\_m512i b, int imm);

VPTERNLOGQ \_\_m512i \_\_mm512\_mask\_ternarylogic\_epi64(\_\_m512i s, \_\_mmask8 m, \_\_m512i a, \_\_m512i b, int imm);

VPTERNLOGQ \_\_m512i \_\_mm512\_maskz\_ternarylogic\_epi64(\_\_mmask8 m, \_\_m512i a, \_\_m512i b, int imm);

VPTERNLOGQ \_\_m256i \_\_mm256\_ternarylogic\_epi64(\_\_m256i a, \_\_m256i b, int imm);

VPTERNLOGQ \_\_m256i \_\_mm256\_mask\_ternarylogic\_epi64(\_\_m256i s, \_\_mmask8 m, \_\_m256i a, \_\_m256i b, int imm);

VPTERNLOGQ \_\_m256i \_\_mm256\_maskz\_ternarylogic\_epi64(\_\_mmask8 m, \_\_m256i a, \_\_m256i b, int imm);

VPTERNLOGQ \_\_m128i \_\_mm\_ternarylogic\_epi64(\_\_m128i a, \_\_m128i b, int imm);

VPTERNLOGQ \_\_m128i \_\_mm\_mask\_ternarylogic\_epi64(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b, int imm);

VPTERNLOGQ \_\_m128i \_\_mm\_maskz\_ternarylogic\_epi64(\_\_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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 26 /r<br>VPTESTMB k2 {k1}, xmm2,<br>xmm3/m128         | A         | V/V                          | AVX512VL<br>AVX512BW     | 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.               |
| EVEX.256.66.0F38.W0 26 /r<br>VPTESTMB k2 {k1}, ymm2,<br>ymm3/m256         | A         | V/V                          | AVX512VL<br>AVX512BW     | 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.               |
| EVEX.512.66.0F38.W0 26 /r<br>VPTESTMB k2 {k1}, zmm2,<br>zmm3/m512         | A         | V/V                          | AVX512BW                 | 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.               |
| EVEX.128.66.0F38.W1 26 /r<br>VPTESTMW k2 {k1}, xmm2,<br>xmm3/m128         | A         | V/V                          | AVX512VL<br>AVX512BW     | 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.               |
| EVEX.256.66.0F38.W1 26 /r<br>VPTESTMW k2 {k1}, ymm2,<br>ymm3/m256         | A         | V/V                          | AVX512VL<br>AVX512BW     | 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.               |
| EVEX.512.66.0F38.W1 26 /r<br>VPTESTMW k2 {k1}, zmm2,<br>zmm3/m512         | A         | V/V                          | AVX512BW                 | 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.               |
| EVEX.128.66.0F38.W0 27 /r<br>VPTESTMD k2 {k1}, xmm2,<br>xmm3/m128/m32bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W0 27 /r<br>VPTESTMD k2 {k1}, ymm2,<br>ymm3/m256/m32bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F38.W0 27 /r<br>VPTESTMD k2 {k1}, zmm2,<br>zmm3/m512/m32bcst | B         | V/V                          | AVX512F                  | 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. |
| EVEX.128.66.0F38.W1 27 /r<br>VPTESTMQ k2 {k1}, xmm2,<br>xmm3/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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.   |
| EVEX.256.66.0F38.W1 27 /r<br>VPTESTMQ k2 {k1}, ymm2,<br>ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F      | 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.   |
| EVEX.512.66.0F38.W1 27 /r<br>VPTESTMQ k2 {k1}, zmm2,<br>zmm3/m512/m64bcst | B         | V/V                          | AVX512F                  | 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.   |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full Mem   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

**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 write-mask. 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[jj] OR \*no writemask\*

    THEN DEST[jj] := (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;

    ELSE DEST[jj] = 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[jj] OR \*no writemask\*

    THEN DEST[jj] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;

    ELSE DEST[jj] = 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[jj] OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[jj] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;

        ELSE DEST[jj] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;

      FI;

    ELSE DEST[jj] := 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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID                | Description  |
|---|-----------|------------------------------|----------------------|--|
| EEX.128.F3.0F38.W0 26 /r<br>VPTESTNMB k2 {k1}, xmm2,<br>xmm3/m128         | A         | V/V                          | AVX512VL<br>AVX512BW | Bitwise NAND of packed byte integers in xmm2 and<br>xmm3/m128 and set mask k2 to reflect the zero/non-zero<br>status of each element of the result, under writemask k1.                  |
| EEX.256.F3.0F38.W0 26 /r<br>VPTESTNMB k2 {k1}, ymm2,<br>ymm3/m256         | A         | V/V                          | AVX512VL<br>AVX512BW | Bitwise NAND of packed byte integers in ymm2 and<br>ymm3/m256 and set mask k2 to reflect the zero/non-zero<br>status of each element of the result, under writemask k1.                  |
| EEX.512.F3.0F38.W0 26 /r<br>VPTESTNMB k2 {k1}, zmm2,<br>zmm3/m512         | A         | V/V                          | AVX512F<br>AVX512BW  | Bitwise NAND of packed byte integers in zmm2 and<br>zmm3/m512 and set mask k2 to reflect the zero/non-zero<br>status of each element of the result, under writemask k1.                  |
| EEX.128.F3.0F38.W1 26 /r<br>VPTESTNMW k2 {k1}, xmm2,<br>xmm3/m128         | A         | V/V                          | AVX512VL<br>AVX512BW | Bitwise NAND of packed word integers in xmm2 and<br>xmm3/m128 and set mask k2 to reflect the zero/non-zero<br>status of each element of the result, under writemask k1.                  |
| EEX.256.F3.0F38.W1 26 /r<br>VPTESTNMW k2 {k1}, ymm2,<br>ymm3/m256         | A         | V/V                          | AVX512VL<br>AVX512BW | Bitwise NAND of packed word integers in ymm2 and<br>ymm3/m256 and set mask k2 to reflect the zero/non-zero<br>status of each element of the result, under writemask k1.                  |
| EEX.512.F3.0F38.W1 26 /r<br>VPTESTNMW k2 {k1}, zmm2,<br>zmm3/m512         | A         | V/V                          | AVX512F<br>AVX512BW  | Bitwise NAND of packed word integers in zmm2 and<br>zmm3/m512 and set mask k2 to reflect the zero/non-zero<br>status of each element of the result, under writemask k1.                  |
| EEX.128.F3.0F38.W0 27 /r<br>VPTESTNMD k2 {k1}, xmm2,<br>xmm3/m128/m32bcst | B         | V/V                          | AVX512VL<br>AVX512F  | Bitwise NAND of packed doubleword integers in xmm2 and<br>xmm3/m128/m32bcst and set mask k2 to reflect the<br>zero/non-zero status of each element of the result, under<br>writemask k1. |
| EEX.256.F3.0F38.W0 27 /r<br>VPTESTNMD k2 {k1}, ymm2,<br>ymm3/m256/m32bcst | B         | V/V                          | AVX512VL<br>AVX512F  | Bitwise NAND of packed doubleword integers in ymm2 and<br>ymm3/m256/m32bcst and set mask k2 to reflect the<br>zero/non-zero status of each element of the result, under<br>writemask k1. |
| EEX.512.F3.0F38.W0 27 /r<br>VPTESTNMD k2 {k1}, zmm2,<br>zmm3/m512/m32bcst | B         | V/V                          | AVX512F              | Bitwise NAND of packed doubleword integers in zmm2 and<br>zmm3/m512/m32bcst and set mask k2 to reflect the<br>zero/non-zero status of each element of the result, under<br>writemask k1. |
| EEX.128.F3.0F38.W1 27 /r<br>VPTESTNMQ k2 {k1}, xmm2,<br>xmm3/m128/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F  | Bitwise NAND of packed quadword integers in xmm2 and<br>xmm3/m128/m64bcst and set mask k2 to reflect the<br>zero/non-zero status of each element of the result, under<br>writemask k1.   |
| EEX.256.F3.0F38.W1 27 /r<br>VPTESTNMQ k2 {k1}, ymm2,<br>ymm3/m256/m64bcst | B         | V/V                          | AVX512VL<br>AVX512F  | Bitwise NAND of packed quadword integers in ymm2 and<br>ymm3/m256/m64bcst and set mask k2 to reflect the<br>zero/non-zero status of each element of the result, under<br>writemask k1.   |
| EEX.512.F3.0F38.W1 27 /r<br>VPTESTNMQ k2 {k1}, zmm2,<br>zmm3/m512/m64bcst | B         | V/V                          | AVX512F              | Bitwise NAND of packed quadword integers in zmm2 and<br>zmm3/m512/m64bcst and set mask k2 to reflect the<br>zero/non-zero status of each element of the result, under<br>writemask k1.   |



## Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full Mem   | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |
| B     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

## Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

## Operation

## VPTESTNMB

(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j := 0 TO KL-1

  i := j\*8

  IF MaskBit(j) OR \*no writemask\*

    THEN

      DEST[j] := (SRC1[j+7:i] BITWISE AND SRC2[j+7:i] == 0)? 1 : 0

    ELSE DEST[j] := 0; zeroing masking only

  FI

ENDFOR

DEST[MAX\_KL-1:KL] := 0

## VPTESTNMW

(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j := 0 TO KL-1

  i := j\*16

  IF MaskBit(j) OR \*no writemask\*

    THEN

      DEST[j] := (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0

    ELSE DEST[j] := 0; zeroing masking only

  FI

ENDFOR

DEST[MAX\_KL-1:KL] := 0

## VPTESTNMD

(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j := 0 TO KL-1

  i := j\*32

  IF MaskBit(j) OR \*no writemask\*

    THEN

      IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

        THEN DEST[i+31:i] := (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0

        ELSE DEST[j] := (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0

    FI

```

        ELSE DEST[j] := 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] := 0

```

**VPTSTNMQ**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j\*64

IF MaskBit(j) OR \*no writemask\*

THEN

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN DEST[j] := (SRC1[i+63:i] BITWISE AND SRC2[63:0] == 0)? 1 : 0;

ELSE DEST[j] := (SRC1[j+63:i] BITWISE AND SRC2[j+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**

```

VPTSTNMB __mmask64 __mm512_testn_epi8_mask(__m512i a, __m512i b);
VPTSTNMB __mmask64 __mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTSTNMB __mmask32 __mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTSTNMB __mmask32 __mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTSTNMB __mmask16 __mm_testn_epi8_mask(__m128i a, __m128i b);
VPTSTNMB __mmask16 __mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTSTNMW __mmask32 __mm512_testn_epi16_mask(__m512i a, __m512i b);
VPTSTNMW __mmask32 __mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTSTNMW __mmask16 __mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTSTNMW __mmask16 __mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTSTNMW __mmask8 __mm_testn_epi16_mask(__m128i a, __m128i b);
VPTSTNMW __mmask8 __mm_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTSTNMD __mmask16 __mm512_testn_epi32_mask(__m512i a, __m512i b);
VPTSTNMD __mmask16 __mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTSTNMD __mmask8 __mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTSTNMD __mmask8 __mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTSTNMD __mmask8 __mm_testn_epi32_mask(__m128i a, __m128i b);
VPTSTNMD __mmask8 __mm_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);
VPTSTNMQ __mmask8 __mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTSTNMQ __mmask8 __mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTSTNMQ __mmask8 __mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTSTNMQ __mmask8 __mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTSTNMQ __mmask8 __mm_testn_epi64_mask(__m128i a, __m128i b);
VPTSTNMQ __mmask8 __mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

```

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

VPTSTNMD/VPTSTNMQ: See Table 2-49, "Type E4 Class Exception Conditions."

VPTSTNMB/VPTSTNMW: See Exceptions Type E4.nb in Table 2-49, "Type E4 Class Exception Conditions."

## VRANGEPD—Range Restriction Calculation for Packed Pairs of Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W1 50 /r ib<br>VRANGEPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst, imm8      | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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.   |
| EVEX.256.66.0F3A.W1 50 /r ib<br>VRANGEPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8      | A         | V/V                          | AVX512VL<br>AVX512DQ     | Calculate four RANGE operation output value from 4pairs 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.   |
| EVEX.512.66.0F3A.W1 50 /r ib<br>VRANGEPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{sae}, imm8 | A         | V/V                          | AVX512DQ                 | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

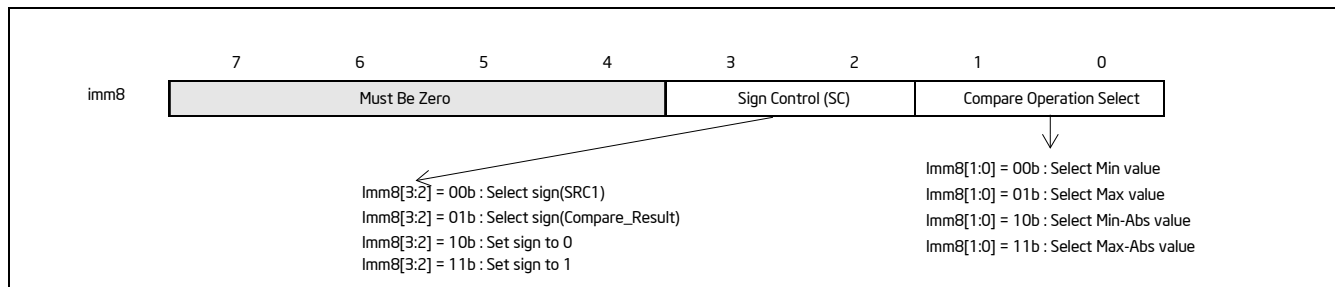


Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-13. 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 5-13.

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 floating-point 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 5-14.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-15.

**Table 5-13. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]**

| Src1  | Src2  | Result       | IE Signaling Due to Comparison | Imm8[3:2] Effect to Range Output |
|-------|-------|--------------|--------------------------------|----------------------------------|
| sNaN1 | sNaN2 | Quiet(sNaN1) | Yes                            | Ignored                          |
| sNaN1 | qNaN2 | Quiet(sNaN1) | Yes                            | Ignored                          |
| sNaN1 | Norm2 | Quiet(sNaN1) | Yes                            | Ignored                          |
| qNaN1 | sNaN2 | Quiet(sNaN2) | Yes                            | Ignored                          |
| qNaN1 | qNaN2 | qNaN1        | No                             | Applicable                       |
| qNaN1 | Norm2 | Norm2        | No                             | Applicable                       |
| Norm1 | sNaN2 | Quiet(sNaN2) | Yes                            | Ignored                          |
| Norm1 | qNaN2 | Norm1        | No                             | Applicable                       |

**Table 5-14. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN\_ABS, and MAX, MAX\_ABS**

| MIN and MIN_ABS |      |        | MAX and MAX_ABS |      |        |
|-----------------|------|--------|-----------------|------|--------|
| Src1            | Src2 | Result | Src1            | Src2 | Result |
| +0              | -0   | -0     | +0              | -0   | +0     |
| -0              | +0   | -0     | -0              | +0   | +0     |

**Table 5-15. 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 5-13
    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 5-14
ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 5-15
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  $\pm 1023$ .

```
VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;
```

Where:

zmm\_dst is the destination operand.

zmm\_src is the input operand to compare against  $\pm 1023$  (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  $|zmm\_src| \geq 1023$ , 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 __mm_range_pd (__m128 a, __m128d b, int imm);
```

```
VRANGEPD __m128d __mm_mask_range_pd (__m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
```

```
VRANGEPD __m128d __mm_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal.

### Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."

## VRANGEPS—Range Restriction Calculation for Packed Pairs of Float32 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W0 50 /r ib<br>VRANGEPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst, imm8      | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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.  |
| EVEX.256.66.0F3A.W0 50 /r ib<br>VRANGEPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8      | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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. |
| EVEX.512.66.0F3A.W0 50 /r ib<br>VRANGEPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{sae}, imm8 | A         | V/V                          | AVX512DQ                 | 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.   |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision floating-point values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-13. 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 5-13.

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 floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPS/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-14.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-15.

**Operation**

```

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-13
    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 5-14
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-15
    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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEEX.LLIG.66.0F3A.W1 51 /r<br>VRANGESD xmm1 {k1}{z},<br>xmm2, xmm3/m64{sae}, imm8 | A         | V/V                          | AVX512DQ                 | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2      | Operand 3     | Operand 4 |
|-------|---------------|---------------|----------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### Description

This instruction calculates a range operation output from two input double precision floating-point values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-13. 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 5-13.

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 floating-point MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGESD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-14.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-15.

**Operation**

```

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-13
    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 5-14
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] := from Table 5-15
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] := (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] := (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest := (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest := TMP[63:0];// Preserve sign of compare result
    10: dest := (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest := (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

**VRANGESD**

```

IF k1[0] OR *no writemask*
    THEN DEST[63:0] := RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
    IF *merging-masking*                ; merging-masking
        THEN *DEST[63:0] remains unchanged*
    ELSE                                  ; zeroing-masking
        DEST[63:0] = 0
FI;
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 1023$ .

```
VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;
```

Where:

xmm\_dst is the destination operand.

xmm\_src is the input operand to compare against  $\pm 1023$ .

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  $|xmm\_src| \geq 1023$ , the selected sign of SRC1 is kept.

Thus, if  $xmm\_src < -1023$ , the result of VRANGESD will be the minimal value of -1023 while if  $xmm\_src > +1023$ , the result of VRANGESD 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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.LLIG.66.0F3A.W0 51 /r<br>VRANGESS xmm1 {k1}{z},<br>xmm2, xmm3/m32{sae}, imm8 | A         | V/V                          | AVX512DQ                 | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction calculates a range operation output from two input single-precision floating-point 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.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of imm8[1:0] and imm8[3:2] are shown in Figure 5-27.

Bits 128:31 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 or more input value is NAN is listed in Table 5-13. 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 5-13.

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 floating-point 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 5-14.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-15.

**Operation**

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```

{
  // Check if SNAN and report IE, see also Table 5-13
  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 5-14
  ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] := from Table 5-15
  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];

```

**VRANGESS**

```

IF k1[0] OR *no writemask*
  THEN DEST[31:0] := RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[31:0] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[31:0] = 0
FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 150$ .

```
VRANGESS 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 VRANGESS 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**

```

VRANGESS __m128 __mm_range_ss ( __m128 a, __m128 b, int imm);
VRANGESS __m128 __mm_range_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 __mm_mask_range_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 __mm_mask_range_round_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 __mm_maskz_range_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 __mm_maskz_range_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal.

**Other Exceptions**

See Table 2-47, "Type E3 Class Exception Conditions."

## VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 4C /r<br>VRCP14PD xmm1 {k1}{z},<br>xmm2/m128/m64bcst | A         | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocals of the packed double precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask. |
| EVEX.256.66.0F38.W1 4C /r<br>VRCP14PD ymm1 {k1}{z},<br>ymm2/m256/m64bcst | A         | V/V                          | AVX512VL<br>AVX512F      | Computes the approximate reciprocals of the packed double precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask. |
| EVEX.512.66.0F38.W1 4C /r<br>VRCP14PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst | A         | V/V                          | AVX512F                  | Computes the approximate reciprocals of the packed double precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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 5-16. VRCP14PD/VRCP14SD Special Cases**

| Input value                 | Result value | Comments                                 |
|-----------------------------|--------------|--|
| $0 \leq X \leq 2^{-1024}$   | INF          | Very small denormal                      |
| $-2^{-1024} \leq X \leq -0$ | -INF         | Very small denormal                      |
| $X > 2^{1022}$              | Underflow    | Up to 18 bits of fractions are returned* |
| $X < -2^{1022}$             | -Underflow   | Up to 18 bits of fractions are returned* |
| $X = 2^{-n}$                | $2^n$        |  |
| $X = -2^{-n}$               | $-2^n$       |  |

\* in this case the mantissa is shifted right by one or two bits

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vcrcp14-vrsqrt14-vcrcp28-vrsqrt28-vexp2>.



**Operation****VRCP14PD ((EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC *is memory*)
      THEN DEST[i+63:i] := APPROXIMATE(1.0/SRC[63:0]);
      ELSE DEST[i+63:i] := APPROXIMATE(1.0/SRC[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] := 0
    FI;
  FI;
ENDFOR;
DEST[MAXVL-1:VL] := 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

VRCP14PD \_\_m512d \_\_mm512\_rcp14\_pd( \_\_m512d a);

VRCP14PD \_\_m512d \_\_mm512\_mask\_rcp14\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a);

VRCP14PD \_\_m512d \_\_mm512\_maskz\_rcp14\_pd( \_\_mmask8 k, \_\_m512d a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions."

## VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

| Opcode/<br>Instruction   | Op<br>/ En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| EVEX.LLIG.66.0F38.W1 4D /r<br>VRCP14SD xmm1 {k1}{z}, xmm2,<br>xmm3/m64 | 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

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e., not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e., correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-16 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at:

<https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### 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/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 4C /r<br>VRCP14PS xmm1 {k1}{z},<br>xmm2/m128/m32bcst | A         | V/V                          | AVX512VL<br>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<br>VRCP14PS ymm1 {k1}{z},<br>ymm2/m256/m32bcst | A         | V/V                          | AVX512VL<br>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<br>VRCP14PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst | A         | V/V                          | AVX512F                  | Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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 5-17. VRCP14PS/VRCP14SS Special Cases**

| Input value                | Result value | Comments   |
|----------------------------|--------------|--|
| $0 \leq X \leq 2^{-128}$   | INF          | Very small denormal                                  |
| $-2^{-128} \leq X \leq -0$ | -INF         | Very small denormal                                  |
| $X > 2^{126}$              | Underflow    | Up to 18 bits of fractions are returned <sup>1</sup> |
| $X < -2^{126}$             | -Underflow   | Up to 18 bits of fractions are returned <sup>1</sup> |
| $X = 2^{-n}$               | $2^n$        |  |
| $X = -2^{-n}$              | $-2^n$       |  |

### NOTES:

1. In this case, the mantissa is shifted right by one or two bits.

A numerically exact implementation of VRCP14xx can be found at:

<https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

**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 \_\_mm512\_mask\_rcp14\_ps( \_\_m256 s, \_\_mmask8 k, \_\_m256 a);

VRCP14PS \_\_m256 \_\_mm512\_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."

## VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.66.0F38.W0 4D /r<br>VRCP14SS xmm1 {k1}{z}, xmm2,<br>xmm3/m32 | A         | V/V                          | AVX512F                  | Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32]. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it 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 5-17 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### 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[127:32] := SRC1[127:32]
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.”

## VRCPPH—Compute Reciprocals of Packed FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|-----------|------------------------------|-------------------------|--|
| EVEX.128.66.MAP6.WO 4C /r<br>VRCPPH xmm1{k1}{z}, xmm2/m128/<br>m16bcst | A         | V/V                          | AVX512-FP16<br>AVX512VL | Compute the approximate reciprocals of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1. |
| EVEX.256.66.MAP6.WO 4C /r<br>VRCPPH ymm1{k1}{z}, ymm2/m256/<br>m16bcst | A         | V/V                          | AVX512-FP16<br>AVX512VL | Compute the approximate reciprocals of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1. |
| EVEX.512.66.MAP6.WO 4C /r<br>VRCPPH zmm1{k1}{z}, zmm2/m512/<br>m16bcst | A         | V/V                          | AVX512-FP16             | Compute the approximate reciprocals of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction performs a SIMD computation of the approximate reciprocals of 8/16/32 packed FP16 values in the source operand (the second operand) and stores the packed FP16 results in the destination operand. The maximum relative error for this approximation is less than  $2^{-11} + 2^{-14}$ .

For special cases, see Table 5-18.

**Table 5-18. VRCPPH/VRCPSH Special Cases**

| Input Value               | Result Value | Comments            |
|---------------------------|--------------|---------------------|
| $0 \leq X \leq 2^{-16}$   | INF          | Very small denormal |
| $-2^{-16} \leq X \leq -0$ | -INF         | Very small denormal |
| $X > +\infty$             | +0           |                     |
| $X < -\infty$             | -0           |                     |
| $X = 2^{-n}$              | $2^n$        |                     |
| $X = -2^{-n}$             | $-2^n$       |                     |

### Operation

**VRCPPH dest{k1}, src**

VL = 128, 256 or 512

KL := VL/16

FOR i := 0 to KL-1:

  IF k1[i] or \*no writemask\*:

    IF SRC is memory and (EVEX.b = 1):

      tsrc := src.fp16[0]

    ELSE:

      tsrc := src.fp16[i]

      DEST.fp16[i] := APPROXIMATE(1.0 / tsrc)

    ELSE IF \*zeroing\*:

      DEST.fp16[i] := 0

    //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0



**Intel C/C++ Compiler Intrinsic Equivalent**

VRCPPH \_\_m128h \_mm\_mask\_rcp\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a);  
 VRCPPH \_\_m128h \_mm\_maskz\_rcp\_ph (\_\_mmask8 k, \_\_m128h a);  
 VRCPPH \_\_m128h \_mm\_rcp\_ph (\_\_m128h a);  
 VRCPPH \_\_m256h \_mm256\_mask\_rcp\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a);  
 VRCPPH \_\_m256h \_mm256\_maskz\_rcp\_ph (\_\_mmask16 k, \_\_m256h a);  
 VRCPPH \_\_m256h \_mm256\_rcp\_ph (\_\_m256h a);  
 VRCPPH \_\_m512h \_mm512\_mask\_rcp\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a);  
 VRCPPH \_\_m512h \_mm512\_maskz\_rcp\_ph (\_\_mmask32 k, \_\_m512h a);  
 VRCPPH \_\_m512h \_mm512\_rcp\_ph (\_\_m512h a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-49, “Type E4 Class Exception Conditions.”

## VRCPSH—Compute Reciprocal of Scalar FP16 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|--|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.66.MAP6.WO 4D /r<br>VRCPSH xmm1{k1}{z}, xmm2, xmm3/<br>m16 | A         | V/V                          | AVX512-FP16           | Compute the approximate reciprocal of the low FP16 value in xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1. Bits 127:16 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  $2^{-11} + 2^{-14}$ .

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

For special cases, see Table 5-18.

### Operation

**VRCPSH dest{k1}, src1, src2**

IF k1[0] or \*no writemask\*:

```
DEST.fp16[0] := APPROXIMATE(1.0 / src2.fp16[0])
```

ELSE IF \*zeroing\*:

```
DEST.fp16[0] := 0
```

//else DEST.fp16[0] remains unchanged

```
DEST[127:16] := src1[127:16]
```

```
DEST[MAXVL-1:128] := 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VRCPSH __m128h __mm_mask_rcp_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
```

```
VRCPSH __m128h __mm_maskz_rcp_sh (__mmask8 k, __m128h a, __m128h b);
```

```
VRCPSH __m128h __mm_rcp_sh (__m128h a, __m128h b);
```

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

EVEX-encoded instruction, see Table 2-58, "Type E10 Class Exception Conditions."

## VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W1 56 /r ib<br>VREDUCEPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512DQ     | Perform reduction transformation on packed double precision floating-point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1. |
| EVEX.256.66.0F3A.W1 56 /r ib<br>VREDUCEPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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. |
| EVEX.512.66.0F3A.W1 56 /r ib<br>VREDUCEPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae},<br>imm8 | A         | V/V                          | AVX512DQ                 | 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.        |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | imm8      | N/A       |

### Description

Perform reduction transformation of the packed binary encoded double precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point 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 floating-point 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<sup>M</sup>", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 <= |\text{Reduced Result}| <= 2^{p-M-1}$

Then if RC ≠ RNE:  $0 <= |\text{Reduced Result}| < 2^{p-M}$

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.

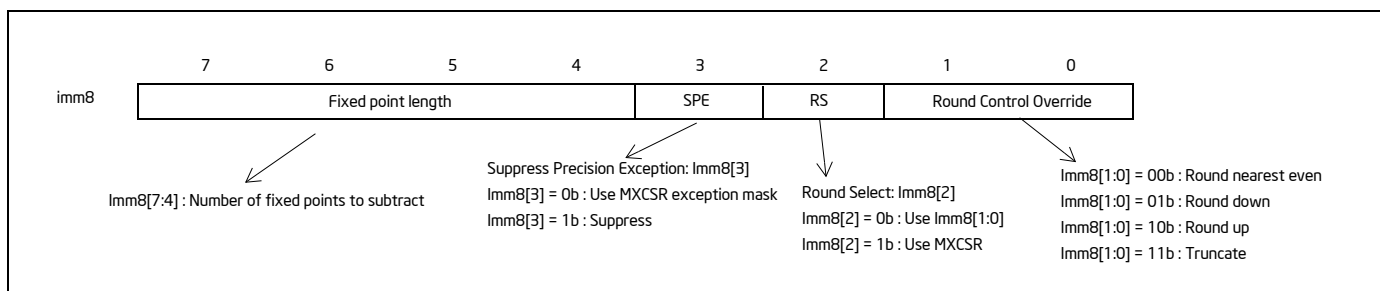


Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-19.

**Table 5-19. VREDUCEPD/SD/PS/SS Special Cases**

|  | Round Mode    | Returned value                  |
|--|---------------|---------------------------------|
| $ \text{Src1}  < 2^{-M-1}$               | RNE           | Src1                            |
| $ \text{Src1}  < 2^{-M}$                 | RPI, Src1 > 0 | Round (Src1-2 <sup>-M</sup> ) * |
|  | RPI, Src1 ≤ 0 | Src1                            |
|  | RNI, Src1 ≥ 0 | Src1                            |
|  | RNI, Src1 < 0 | Round (Src1+2 <sup>-M</sup> ) * |
| Src1 = ±0, or<br>Dest = ±0 (Src1! = INF) | NOT RNI       | +0.0                            |
|  | RNI           | -0.0                            |
| Src1 = ±INF                              | any           | +0.0                            |
| Src1 = ±NAN                              | n/a           | QNaN(Src1)                      |

\* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

### Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); FI;
  M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC := imm8[1:0]; // Round Control for ROUND() operation
  RC source := imm[2];
  SPE := imm[3]; // Suppress Precision Exception
  TMP[63:0] := 2-M * ROUND(2M * SRC[63:0], SPE, RC_source, RC); // ROUND() treats SRC and 2M as standard binary FP values
  TMP[63:0] := SRC[63:0] - TMP[63:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

### VREDUCEPD

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

  i := j \* 64

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b == 1) AND (SRC \*is memory\*)

      THEN DEST[i+63:i] := ReduceArgumentDP(SRC[63:0], imm8[7:0]);

      ELSE DEST[i+63:i] := ReduceArgumentDP(SRC[i+63:i], imm8[7:0]);

    FI;

  ELSE

    IF \*merging-masking\* ; merging-masking

      THEN \*DEST[i+63:i] remains unchanged\*

    ELSE ; zeroing-masking

      DEST[i+63:i] = 0

    FI;

  FI;

ENDFOR;

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCEPD \_\_m512d \_\_mm512\_mask\_reduce\_pd( \_\_m512d a, int imm, int sae)  
 VREDUCEPD \_\_m512d \_\_mm512\_mask\_reduce\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int imm, int sae)  
 VREDUCEPD \_\_m512d \_\_mm512\_maskz\_reduce\_pd(\_\_mmask8 k, \_\_m512d a, int imm, int sae)  
 VREDUCEPD \_\_m256d \_\_mm256\_mask\_reduce\_pd( \_\_m256d a, int imm)  
 VREDUCEPD \_\_m256d \_\_mm256\_mask\_reduce\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, int imm)  
 VREDUCEPD \_\_m256d \_\_mm256\_maskz\_reduce\_pd(\_\_mmask8 k, \_\_m256d a, int imm)  
 VREDUCEPD \_\_m128d \_\_mm\_mask\_reduce\_pd( \_\_m128d a, int imm)  
 VREDUCEPD \_\_m128d \_\_mm\_mask\_reduce\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, int imm)  
 VREDUCEPD \_\_m128d \_\_mm\_maskz\_reduce\_pd(\_\_mmask8 k, \_\_m128d a, int imm)

**SIMD Floating-Point Exceptions**

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”

Additionally:

#UD                      If EVEX.vvvv != 1111B.

## VREDUCEPH—Perform Reduction Transformation on Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.OF3A.W0 56 /r /ib<br>VREDUCEPH xmm1{k1}{z}, xmm2/<br>m128/m16bcst, imm8       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Perform reduction transformation on packed FP16 values in xmm2/m128/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. |
| EVEX.256.NP.OF3A.W0 56 /r /ib<br>VREDUCEPH ymm1{k1}{z}, ymm2/<br>m256/m16bcst, imm8       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Perform reduction transformation on packed FP16 values in ymm2/m256/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in ymm1 subject to writemask k1. |
| EVEX.512.NP.OF3A.W0 56 /r /ib<br>VREDUCEPH zmm1{k1}{z}, zmm2/<br>m512/m16bcst {sae}, imm8 | A         | V/V                          | AVX512-FP16             | Perform reduction transformation on packed FP16 values in zmm2/m512/m16bcst by subtracting a number of fraction bits specified by the imm8 field. Store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | imm8 (r)  | N/A       |

### Description

This instruction performs a reduction transformation of the packed binary encoded FP16 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]. 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 RC=RNE:  $0 \leq |\text{ReducedResult}| \leq 2^{-M-1}$ .

Then if RC  $\neq$  RNE:  $0 \leq |\text{ReducedResult}| < 2^{-M}$ .

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.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 5-20.

Table 5-20. VREDUCEPH/VREDUCESH Special Cases

| Input value                          | Round Mode   | Returned Value                         |
|--------------------------------------|--------------|--|
| $ \text{Src1}  < 2^{-M-1}$           | RNE          | Src1                                   |
| $ \text{Src1}  < 2^{-M}$             | RU, Src1 > 0 | $\text{Round}(\text{Src1} - 2^{-M})^1$ |
|                                      | RU, Src1 ≤ 0 | Src1                                   |
|                                      | RD, Src1 ≥ 0 | Src1                                   |
|                                      | RD, Src1 < 0 | $\text{Round}(\text{Src1} + 2^{-M})$   |
| Src1 = ±0 or<br>Dest = ±0 (Src1 ≠ ∞) | NOT RD       | +0.0                                   |
|                                      | RD           | -0.0                                   |
| Src1 = ±∞                            | Any          | +0.0                                   |
| Src1 = ±NaN                          | Any          | QNaN (Src1)                            |

**NOTES:**

1. The Round(.) function uses rounding controls specified by (imm8[2]? MXCSR.RC: imm8[1:0]).

**Operation**

```
def reduce_fp16(src, imm8):
    nan := (src.exp = 0x1F) and (src.fraction != 0)
    if nan:
        return QNaN(src)
    m := imm8[7:4]
    rc := imm8[1:0]
    rc_source := imm8[2]
    spe := imm8[3] // suppress precision exception
    tmp := 2-(m) * ROUND(2m * src, spe, rc_source, rc)
    tmp := src - tmp // using same RC, SPE controls
    return tmp
```

**VREDUCEPH dest{k1}, src, imm8**

VL = 128, 256 or 512

KL := VL/16

FOR i := 0 to KL-1:

IF k1[i] or \*no writemask\*:

IF SRC is memory and (EVEX.b = 1):

tsrc := src.fp16[0]

ELSE:

tsrc := src.fp16[i]

DEST.fp16[i] := reduce\_fp16(tsrc, imm8)

ELSE IF \*zeroing\*:

DEST.fp16[i] := 0

//else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VREDUCEPH \_\_m128h \_mm\_mask\_reduce\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, int imm8);  
 VREDUCEPH \_\_m128h \_mm\_maskz\_reduce\_ph (\_\_mmask8 k, \_\_m128h a, int imm8);  
 VREDUCEPH \_\_m128h \_mm\_reduce\_ph (\_\_m128h a, int imm8);  
 VREDUCEPH \_\_m256h \_mm256\_mask\_reduce\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a, int imm8);  
 VREDUCEPH \_\_m256h \_mm256\_maskz\_reduce\_ph (\_\_mmask16 k, \_\_m256h a, int imm8);  
 VREDUCEPH \_\_m256h \_mm256\_reduce\_ph (\_\_m256h a, int imm8);  
 VREDUCEPH \_\_m512h \_mm512\_mask\_reduce\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, int imm8);  
 VREDUCEPH \_\_m512h \_mm512\_maskz\_reduce\_ph (\_\_mmask32 k, \_\_m512h a, int imm8);  
 VREDUCEPH \_\_m512h \_mm512\_reduce\_ph (\_\_m512h a, int imm8);  
 VREDUCEPH \_\_m512h \_mm512\_mask\_reduce\_round\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, int imm8, const int sae);  
 VREDUCEPH \_\_m512h \_mm512\_maskz\_reduce\_round\_ph (\_\_mmask32 k, \_\_m512h a, int imm8, const int sae);  
 VREDUCEPH \_\_m512h \_mm512\_reduce\_round\_ph (\_\_m512h a, int imm8, const int sae);

**SIMD Floating-Point Exceptions**

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”



## VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F3A.W0 56 /r ib<br>VREDUCEPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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. |
| EVEX.256.66.0F3A.W0 56 /r ib<br>VREDUCEPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8         | A         | V/V                          | AVX512VL<br>AVX512DQ     | 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. |
| EVEX.512.66.0F3A.W0 56 /r ib<br>VREDUCEPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{sae},<br>imm8 | A         | V/V                          | AVX512DQ                 | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | imm8      | N/A       |

#### Description

Perform reduction transformation of the packed binary encoded single-precision floating-point values in the source operand (the second operand) and store the reduced results in binary floating-point 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 floating-point 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<sup>M</sup>", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-19.

**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(2M * SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M 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.

## VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

| Opcode/<br>Instruction   | Op /<br>En | 64/32 bit<br>Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|------------|------------------------------|--------------------------|---|
| EVEX.LLIG.66.0F3A.W1 57<br>VREDUCESD xmm1 {k1}{z},<br>xmm2, xmm3/m64{sae},<br>imm8/r | A          | V/V                          | AVX512D<br>Q             | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Perform a reduction transformation of the binary encoded double precision floating-point value in the low qword element of the second source operand (the third operand) and store the reduced result in binary floating-point 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 floating-point 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<sup>M</sup>", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 <= |\text{Reduced Result}| <= 2^{p-M-1}$

Then if RC ≠ RNE:  $0 <= |\text{Reduced Result}| < 2^{p-M}$

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.

The operation is write masked.

Handling of special case of input values are listed in Table 5-19.

### Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [63:0] = NAN) THEN
    RETURN (Convert SRC[63:0] to QNaN); FI;
  M := imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC := imm8[1:0]; // Round Control for ROUND() operation
  RC_source := imm[2];
  SPE := imm[3]; // Suppress Precision Exception
  TMP[63:0] := 2-M * {ROUND(2M*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M 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."

## VREDUCESH—Perform Reduction Transformation on Scalar FP16 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.NP.OF3A.W0 57 /r /ib<br>VREDUCESH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {sae}, imm8 | A         | V/V                          | AVX512-FP16           | Perform a reduction transformation on the low binary encoded FP16 value in xmm3/m16 by subtracting a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | imm8 (r)  |

### Description

This instruction performs a reduction transformation of the low binary encoded FP16 value in the source operand (the second operand) and store the reduced result in binary FP format to the low element of the destination operand (the first operand) under the writemask k1. For further details see the description of VREDUCEPH.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

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.

This instruction may generate tiny non-zero result. If it does so, it does not report underflow exception, even if underflow exceptions are unmasked (UM flag in MXCSR register is 0).

For special cases, see Table 5-20.

### Operation

#### VREDUCESH dest{k1}, src, imm8

IF k1[0] or \*no writemask\*:

```
dest.fp16[0] := reduce_fp16(src2.fp16[0], imm8) // see VREDUCEPH
```

ELSE IF \*zeroing\*:

```
dest.fp16[0] := 0
```

//else dest.fp16[0] remains unchanged

```
DEST[127:16] := src1[127:16]
```

```
DEST[MAXVL-1:128] := 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VREDUCESH __m128h __mm_mask_reduce_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
```

```
VREDUCESH __m128h __mm_maskz_reduce_round_sh (__mmask8 k, __m128h a, __m128h b, int imm8, const int sae);
```

```
VREDUCESH __m128h __mm_reduce_round_sh (__m128h a, __m128h b, int imm8, const int sae);
```

```
VREDUCESH __m128h __mm_mask_reduce_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int imm8);
```

```
VREDUCESH __m128h __mm_maskz_reduce_sh (__mmask8 k, __m128h a, __m128h b, int imm8);
```

```
VREDUCESH __m128h __mm_reduce_sh (__m128h a, __m128h b, int imm8);
```

### SIMD Floating-Point Exceptions

Invalid, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

## VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bitMode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-----------|-----------------------------|--------------------------|--|
| EVEX.LLIG.66.0F3A.W0 57 /r<br>/ib<br>VREDUCESS xmm1 {k1}{z},<br>xmm2, xmm3/m32{sae},<br>imm8 | A         | V/V                         | AVX512DQ                 | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Perform a reduction transformation of the binary encoded single-precision floating-point value in the low dword element of the second source operand (the third operand) and store the reduced result in binary floating-point 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 floating-point 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<sup>M</sup>", and their product as binary floating-point numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , where 'man<sub>2</sub>' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE:  $0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$

Then if RC ≠ RNE:  $0 \leq |\text{Reduced Result}| < 2^{p-M}$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e., Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-19.

### 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(2M*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2M as standard binary FP values
  TMP[31:0] := SRC[31:0] - TMP[31:0]; // subtraction under the same RC,SPE controls
  RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

**VREDUCESS**

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] := ReduceArgumentSP(SRC2[31:0], imm8[7:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] = 0
    FI;
FI;
DEST[127:32] := SRC1[127:32]
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

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W1 09 /r ib<br>VRNDSCALEPD xmm1 {k1}{z},<br>xmm2/m128/m64bcst, imm8      | A         | V/V                          | AVX512VL<br>AVX512F      | 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.   |
| EVEX.256.66.0F3A.W1 09 /r ib<br>VRNDSCALEPD ymm1 {k1}{z},<br>ymm2/m256/m64bcst, imm8      | A         | V/V                          | AVX512VL<br>AVX512F      | 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.   |
| EVEX.512.66.0F3A.W1 09 /r ib<br>VRNDSCALEPD zmm1 {k1}{z},<br>zmm2/m512/m64bcst{sae}, imm8 | A         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | imm8      | N/A       |

#### 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

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

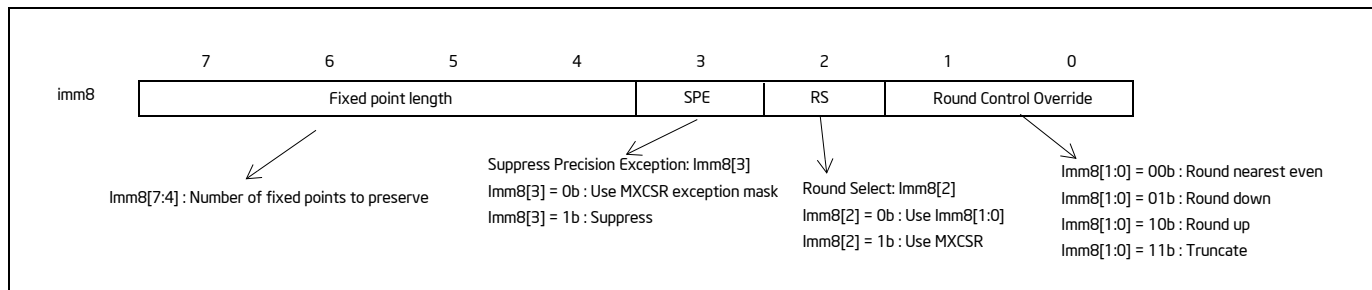


Figure 5-29. Imm8 Controls for VRNDSCALEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-21.

Table 5-21. VRNDSCALEPD/SD/PS/SS Special Cases

|                  | Returned value         |
|------------------|------------------------|
| <b>Src1=±inf</b> | Src1                   |
| <b>Src1=±NAN</b> | Src1 converted to QNAN |
| <b>Src1=±0</b>   | Src1                   |

**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(2M*SRC[63:0])
  01: TMP[63:0] := round_to_equal_or_smaller_integer(2M*SRC[63:0])
  10: TMP[63:0] := round_to_equal_or_larger_integer(2M*SRC[63:0])
  11: TMP[63:0] := round_to_nearest_smallest_magnitude_integer(2M*SRC[63:0])
  ESAC

  Dest[63:0] := 2-M* TMP[63:0] ; scale down back to 2-M

  if (imm8[3] = 0) Then ; check SPE
    if (SRC[63:0] != Dest[63:0]) Then ; check precision lost
      set_precision() ; set #PE
    FI;
  FI;
}
    
```

```

    return(Dest[63:0])
}

```

### VRNDSCALEPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

IF \*src is a memory operand\*

THEN TMP\_SRC := BROADCAST64(SRC, VL, k1)

ELSE TMP\_SRC := SRC

FI;

FOR j := 0 TO KL-1

i := j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] := RoundToIntegerDP((TMP\_SRC[i+63:i], imm8[7:0])

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] := 0

### Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPD \_\_m512d \_\_mm512\_roundscale\_pd( \_\_m512d a, int imm);

VRNDSCALEPD \_\_m512d \_\_mm512\_roundscale\_round\_pd( \_\_m512d a, int imm, int sae);

VRNDSCALEPD \_\_m512d \_\_mm512\_mask\_roundscale\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int imm);

VRNDSCALEPD \_\_m512d \_\_mm512\_mask\_roundscale\_round\_pd(\_\_m512d s, \_\_mmask8 k, \_\_m512d a, int imm, int sae);

VRNDSCALEPD \_\_m512d \_\_mm512\_maskz\_roundscale\_pd( \_\_mmask8 k, \_\_m512d a, int imm);

VRNDSCALEPD \_\_m512d \_\_mm512\_maskz\_roundscale\_round\_pd( \_\_mmask8 k, \_\_m512d a, int imm, int sae);

VRNDSCALEPD \_\_m256d \_\_mm256\_roundscale\_pd( \_\_m256d a, int imm);

VRNDSCALEPD \_\_m256d \_\_mm256\_mask\_roundscale\_pd(\_\_m256d s, \_\_mmask8 k, \_\_m256d a, int imm);

VRNDSCALEPD \_\_m256d \_\_mm256\_maskz\_roundscale\_pd( \_\_mmask8 k, \_\_m256d a, int imm);

VRNDSCALEPD \_\_m128d \_\_mm\_roundscale\_pd( \_\_m128d a, int imm);

VRNDSCALEPD \_\_m128d \_\_mm\_mask\_roundscale\_pd(\_\_m128d s, \_\_mmask8 k, \_\_m128d a, int imm);

VRNDSCALEPD \_\_m128d \_\_mm\_maskz\_roundscale\_pd( \_\_mmask8 k, \_\_m128d a, int imm);

### SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

### Other Exceptions

See Table 2-46, "Type E2 Class Exception Conditions."

## VRNDSCALEPH—Round Packed FP16 Values to Include a Given Number of Fraction Bits

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.NP.OF3A.W0 08 /r /ib<br>VRNDSCALEPH xmm1{k1}{z}, xmm2/<br>m128/m16bcst, imm8       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Round packed FP16 values in xmm2/m128/<br>m16bcst to a number of fraction bits specified by<br>the imm8 field. Store the result in xmm1 subject<br>to writemask k1. |
| EVEX.256.NP.OF3A.W0 08 /r /ib<br>VRNDSCALEPH ymm1{k1}{z}, ymm2/<br>m256/m16bcst, imm8       | A         | V/V                          | AVX512-FP16<br>AVX512VL | Round packed FP16 values in ymm2/m256/<br>m16bcst to a number of fraction bits specified by<br>the imm8 field. Store the result in ymm1 subject<br>to writemask k1. |
| EVEX.512.NP.OF3A.W0 08 /r /ib<br>VRNDSCALEPH zmm1{k1}{z}, zmm2/<br>m512/m16bcst {sae}, imm8 | A         | V/V                          | AVX512-FP16             | Round packed FP16 values in zmm2/m512/<br>m16bcst to a number of fraction bits specified by<br>the imm8 field. Store the result in zmm1 subject<br>to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | imm8 (r)  | N/A       |

#### Description

This instruction rounds the FP16 values in the source operand by the rounding mode specified in the immediate operand (see Table 5-22) and places the result in the destination operand. The destination operand is conditionally updated according to the writemask.

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 an FP16 value.

Note 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 Table 5-22, “Imm8 Controls for VRNDSCALEPH/VRNDSCALESH.” Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

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.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 5-23.

The formula of the operation on each data element for VRNDSCALEPH is

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

If this instruction encoding’s SPE bit (bit 3) in the immediate operand is 1, VRNDSCALEPH can set MXCSR.UE without MXCSR.PE.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Table 5-22. Imm8 Controls for VRNDSCALEPH/VRNDSCALESH

| Imm8 Bits | Description   |
|-----------|---|
| imm8[7:4] | Number of fixed points to preserve.   |
| imm8[3]   | Suppress Precision Exception (SPE)<br>0b00: Implies use of MXCSR exception mask.<br>0b01: Implies suppress.     |
| imm8[2]   | Round Select (RS)<br>0b00: Implies use of imm8[1:0].<br>0b01: Implies use of MXCSR.                             |
| imm8[1:0] | Round Control Override:<br>0b00: Round nearest even.<br>0b01: Round down.<br>0b10: Round up.<br>0b11: Truncate. |

Table 5-23. VRNDSCALEPH/VRNDSCALESH Special Cases

| Input Value            | Returned Value         |
|------------------------|------------------------|
| Src1 = $\pm\infty$     | Src1                   |
| Src1 = $\pm\text{NaN}$ | Src1 converted to QNaN |
| Src1 = $\pm 0$         | Src1                   |

### Operation

```

def round_fp16_to_integer(src, imm8):
    if imm8[2] = 1:
        rounding_direction := MXCSR.RC
    else:
        rounding_direction := imm8[1:0]
    m := imm8[7:4] // scaling factor

    trc1 := 2^m * src

    if rounding_direction = 0b00:
        tmp := round_to_nearest_even_integer(trc1)
    else if rounding_direction = 0b01:
        tmp := round_to_equal_or_smaller_integer(trc1)
    else if rounding_direction = 0b10:
        tmp := round_to_equal_or_larger_integer(trc1)
    else if rounding_direction = 0b11:
        tmp := round_to_smallest_magnitude_integer(trc1)

    dst := 2^(-m) * tmp

    if imm8[3]==0: // check SPE
        if src != dst:
            MXCSR.PE := 1
    return dst

```

**VRNDSCALEPH dest{k1}, src, imm8**

VL = 128, 256 or 512

KL := VL/16

FOR i := 0 to KL-1:

IF k1[i] or \*no writemask\*:

IF SRC is memory and (EVEX.b = 1):

tsrc := src.fp16[0]

ELSE:

tsrc := src.fp16[i]

DEST.fp16[i] := round\_fp16\_to\_integer(tsrc, imm8)

ELSE IF \*zeroing\*:

DEST.fp16[i] := 0

//else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRNDSCALEPH \_\_m128h \_\_mm\_mask\_roundscale\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, int imm8);

VRNDSCALEPH \_\_m128h \_\_mm\_maskz\_roundscale\_ph (\_\_mmask8 k, \_\_m128h a, int imm8);

VRNDSCALEPH \_\_m128h \_\_mm\_roundscale\_ph (\_\_m128h a, int imm8);

VRNDSCALEPH \_\_m256h \_\_mm256\_mask\_roundscale\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a, int imm8);

VRNDSCALEPH \_\_m256h \_\_mm256\_maskz\_roundscale\_ph (\_\_mmask16 k, \_\_m256h a, int imm8);

VRNDSCALEPH \_\_m256h \_\_mm256\_roundscale\_ph (\_\_m256h a, int imm8);

VRNDSCALEPH \_\_m512h \_\_mm512\_mask\_roundscale\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, int imm8);

VRNDSCALEPH \_\_m512h \_\_mm512\_maskz\_roundscale\_ph (\_\_mmask32 k, \_\_m512h a, int imm8);

VRNDSCALEPH \_\_m512h \_\_mm512\_roundscale\_ph (\_\_m512h a, int imm8);

VRNDSCALEPH \_\_m512h \_\_mm512\_mask\_roundscale\_round\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, int imm8, const int sae);

VRNDSCALEPH \_\_m512h \_\_mm512\_maskz\_roundscale\_round\_ph (\_\_mmask32 k, \_\_m512h a, int imm8, const int sae);

VRNDSCALEPH \_\_m512h \_\_mm512\_roundscale\_round\_ph (\_\_m512h a, int imm8, const int sae);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Precision.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-46, "Type E2 Class Exception Conditions."

## VRNDSCALEPS—Round Packed Float32 Values to Include a Given Number of Fraction Bits

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.128.66.0F3A.W0 08 /r ib<br>VRNDSCALEPS xmm1 {k1}{z},<br>xmm2/m128/m32bcst, imm8      | A         | V/V                          | AVX512VL<br>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<br>VRNDSCALEPS ymm1 {k1}{z},<br>ymm2/m256/m32bcst, imm8      | A         | V/V                          | AVX512VL<br>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<br>VRNDSCALEPS zmm1 {k1}{z},<br>zmm2/m512/m32bcst{sae}, imm8 | 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

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | imm8      | N/A       |

#### Description

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.  
Handling of special case of input values are listed in Table 5-21.

### 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(2M*SRC[31:0])
  01: TMP[31:0] := round_to_equal_or_smaller_integer(2M*SRC[31:0])
  10: TMP[31:0] := round_to_equal_or_larger_integer(2M*SRC[31:0])
  11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer(2M*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 __mm_roundscale_ps( __m256 a, int imm);
VRNDSCALEPS __m128 __mm_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 __mm_maskz_roundscale_ps( __mmask8 k, __m128 a, int imm);

```

### SIMD Floating-Point Exceptions

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

### Other Exceptions

See Table 2-46, “Type E2 Class Exception Conditions.”

## VRNDSCALESD—Round Scalar Float64 Value to Include a Given Number of Fraction Bits

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.LLIG.66.0F3A.W1 0B /r ib<br>VRNDSCALESD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{sae}, imm8 | A         | V/V                          | AVX512F                  | 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

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | imm8      |

### 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 VRNDSCALESD is

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-21.

**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(2M*SRC[63:0])
  01: TMP[63:0] := round_to_equal_or_smaller_integer(2M*SRC[63:0])
  10: TMP[63:0] := round_to_equal_or_larger_integer(2M*SRC[63:0])
  11: TMP[63:0] := round_to_nearest_smallest_magnitude_integer(2M*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."

## VRNDSCALESH—Round Scalar FP16 Value to Include a Given Number of Fraction Bits

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|---|-----------|------------------------------|-----------------------|--|
| EVEEX.LLIG.NP.OF3A.W0 0A /r /ib<br>VRNDSCALESH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {sae}, imm8 | A         | V/V                          | AVX512-FP16           | Round the low FP16 value in xmm3/m16 to a number of fraction bits specified by the imm8 field. Store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | imm8 (r)  |

### Description

This instruction rounds the low FP16 value in the second source operand by the rounding mode specified in the immediate operand (see Table 5-22) and places the result in the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

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 FP16 value.

Note 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 Table 5-22, "Imm8 Controls for VRNDSCALEPH/VRNDSCALESH." Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control, and bits 1:0 specify a non-sticky rounding-mode value.

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.

The sign of the result of this instruction is preserved, including the sign of zero. Special cases are described in Table 5-23.

If this instruction encoding's SPE bit (bit 3) in the immediate operand is 1, VRNDSCALESH can set MXCSR.UE without MXCSR.PE.

The formula of the operation on each data element for VRNDSCALESH is:

$$\begin{aligned} \text{ROUND}(x) &= 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \\ M &= \text{imm}[7:4]; \end{aligned}$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

### Operation

**VRNDSCALESH dest{k1}, src1, src2, imm8**

IF k1[0] or \*no writemask\*:

```
DEST.fp16[0] := round_fp16_to_integer(src2.fp16[0], imm8) // see VRNDSCALEPH
```

ELSE IF \*zeroing\*:

```
DEST.fp16[0] := 0
```

//else DEST.fp16[0] remains unchanged

```
DEST[127:16] = src1[127:16]
```

```
DEST[MAXVL-1:128] := 0
```

**Intel C/C++ Compiler Intrinsic Equivalent**

VRNDSCALESH \_\_m128h \_\_mm\_mask\_roundscale\_round\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b, int imm8, const int sae);

VRNDSCALESH \_\_m128h \_\_mm\_maskz\_roundscale\_round\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, int imm8, const int sae);

VRNDSCALESH \_\_m128h \_\_mm\_roundscale\_round\_sh (\_\_m128h a, \_\_m128h b, int imm8, const int sae);

VRNDSCALESH \_\_m128h \_\_mm\_mask\_roundscale\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b, int imm8);

VRNDSCALESH \_\_m128h \_\_mm\_maskz\_roundscale\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, int imm8);

VRNDSCALESH \_\_m128h \_\_mm\_roundscale\_sh (\_\_m128h a, \_\_m128h b, int imm8);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Precision.

**Other Exceptions**

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

## VRNDSCALESS—Round Scalar Float32 Value to Include a Given Number of Fraction Bits

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|------------|------------------------------|--------------------------|---|
| EVEX.LLIG.66.0F3A.W0 0A /r ib<br>VRNDSCALESS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{sae}, imm8 | A          | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### 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

$$\text{ROUND}(x) = 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

$$M = \text{imm}[7:4];$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e., no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

$$\text{ROUND}(x) = \text{Round\_to\_INT}(x, \text{round\_ctrl}),$$

$$\text{round\_ctrl} = \text{imm}[3:0];$$

EVEX encoded version: The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-21.

**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(2M*SRC[31:0])
    01: TMP[31:0] := round_to_equal_or_smaller_integer(2M*SRC[31:0])
    10: TMP[31:0] := round_to_equal_or_larger_integer(2M*SRC[31:0])
    11: TMP[31:0] := round_to_nearest_smallest_magnitude_integer(2M*SRC[31:0])
    ESAC;

    Dest[31:0] := 2-M* TMP[31:0]           ; scale down back to 2-M
    if (imm8[3] = 0) Then                   ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then   ; check precision lost
            set_precision()                 ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

```

VRNDSCALESS (EVEX encoded version)

```

IF k1[0] or *no writemask*
    THEN  DEST[31:0] := RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE                                  ; zeroing-masking
            THEN DEST[31:0] := 0
        FI;
    FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VRNDSCALESS __m128 __mm_roundscale_ss ( __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 __mm_roundscale_round_ss ( __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 __mm_mask_roundscale_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 __mm_mask_roundscale_round_ss ( __m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 __mm_maskz_roundscale_ss ( __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 __mm_maskz_roundscale_round_ss ( __mmask8 k, __m128 a, __m128 b, int imm, int sae);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision.

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

**Other Exceptions**

See Table 2-47, “Type E3 Class Exception Conditions.”

## VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W1 4E /r<br>VRSQRT14PD xmm1 {k1}{z},<br>xmm2/m128/m64bcst | A         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W1 4E /r<br>VRSQRT14PD ymm1 {k1}{z},<br>ymm2/m256/m64bcst | A         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F38.W1 4E /r<br>VRSQRT14PD zmm1 {k1}{z},<br>zmm2/m512/m64bcst | A         | V/V                          | AVX512F                  | 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.  |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.



**Operation****VRSQRT14PD (EVEX encoded versions)**

(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j \* 64

IF k1[j] OR \*no writemask\* THEN

IF (EVEX.b = 1) AND (SRC \*is memory\*)

THEN DEST[i+63:i] := APPROXIMATE(1.0/ SQRT(SRC[63:0]));

ELSE DEST[i+63:i] := APPROXIMATE(1.0/ SQRT(SRC[i+63:i]));

FI;

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE ; zeroing-masking

DEST[i+63:i] := 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] := 0

**Table 5-24. VRSQRT14PD Special Cases**

| Input value   | Result value    | Comments                 |
|---------------|-----------------|--------------------------|
| Any denormal  | Normal          | Cannot generate overflow |
| $X = 2^{-2n}$ | $2^n$           |                          |
| $X < 0$       | QNaN_Indefinite | Including -INF           |
| $X = -0$      | -INF            |                          |
| $X = +0$      | +INF            |                          |
| $X = +INF$    | +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 \_\_mm128\_rsqrt14\_pd( \_\_m128d a);

VRSQRT14PD \_\_m128d \_\_mm128\_mask\_rsqrt14\_pd( \_\_m128d s, \_\_mmask8 k, \_\_m128d a);

VRSQRT14PD \_\_m128d \_\_mm128\_maskz\_rsqrt14\_pd( \_\_mmask8 k, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-49, "Type E4 Class Exception Conditions."

## VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.LLIG.66.0F38.W1 4F /r<br>VRSQRT14SD xmm1 {k1}{z},<br>xmm2, xmm3/m64 | A         | V/V                          | AVX512F                  | 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]. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Computes the approximate reciprocal of the square roots of the scalar double precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT14SD (EVEX version)

IF k1[0] or \*no writemask\*

THEN DEST[63:0] := APPROXIMATE(1.0/ SQRT(SRC2[63:0]))

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] := 0

FI;

FI;

DEST[127:64] := SRC1[127:64]

DEST[MAXVL-1:128] := 0

**Table 5-25. VRSQRT14SD Special Cases**

| Input value   | Result value    | Comments                 |
|---------------|-----------------|--------------------------|
| Any denormal  | Normal          | Cannot generate overflow |
| $X = 2^{-2n}$ | $2^n$           |                          |
| $X < 0$       | QNaN_Indefinite | Including -INF           |
| $X = -0$      | -INF            |                          |
| $X = +0$      | +INF            |                          |
| $X = +INF$    | +0              |                          |

**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

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 4E /r<br>VRSQRT14PS xmm1 {k1}{z},<br>xmm2/m128/m32bcst | A         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.256.66.0F38.W0 4E /r<br>VRSQRT14PS ymm1 {k1}{z},<br>ymm2/m256/m32bcst | A         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F38.W0 4E /r<br>VRSQRT14PS zmm1 {k1}{z},<br>zmm2/m512/m32bcst | A         | V/V                          | AVX512F                  | 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. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|------------|---------------|---------------|-----------|-----------|
| A     | Full       | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### 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  $\infty$  with the sign of the source value is returned. When the source operand is an  $+\infty$  then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN\_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

**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

**Table 5-26. VRSQRT14PS Special Cases**

| Input value   | Result value    | Comments                 |
|---------------|-----------------|--------------------------|
| Any denormal  | Normal          | Cannot generate overflow |
| $X = 2^{-2n}$ | $2^n$           |                          |
| $X < 0$       | QNaN_Indefinite | Including -INF           |
| $X = -0$      | -INF            |                          |
| $X = +0$      | +INF            |                          |
| $X = +INF$    | +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."

## VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.LLIG.66.0F38.W0 4F /r<br>VRSQRT14SS xmm1 {k1}{z},<br>xmm2, xmm3/m32 | A         | V/V                          | AVX512F                  | 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]. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|---------------|---------------|--------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Computes the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $\infty$ , zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at <https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2>.

### Operation

#### VRSQRT14SS (EVEX version)

```

IF k1[0] or *no writemask*
  THEN  DEST[31:0] := APPROXIMATE(1.0/ SQRT(SRC2[31:0]))
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      THEN DEST[31:0] := 0
  FI;
FI;
DEST[127:32] := SRC1[127:32]
DEST[MAXVL-1:128] := 0

```

**Table 5-27. VRSQRT14SS Special Cases**

| Input value   | Result value    | Comments                 |
|---------------|-----------------|--------------------------|
| Any denormal  | Normal          | Cannot generate overflow |
| $X = 2^{-2n}$ | $2^n$           |                          |
| $X < 0$       | QNaN_Indefinite | Including -INF           |
| $X = -0$      | -INF            |                          |
| $X = +0$      | +INF            |                          |
| $X = +INF$    | +0              |                          |

**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."

## VRSQRTPH—Compute Reciprocals of Square Roots of Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EEX.128.66.MAP6.WO 4E /r<br>VRSQRTPH xmm1{k1}{z}, xmm2/<br>m128/m16bcst | A         | V/V                          | AVX512-FP16<br>AVX512VL | Compute the approximate reciprocals of the square roots of packed FP16 values in xmm2/m128/m16bcst and store the result in xmm1 subject to writemask k1. |
| EEX.256.66.MAP6.WO 4E /r<br>VRSQRTPH ymm1{k1}{z}, ymm2/<br>m256/m16bcst | A         | V/V                          | AVX512-FP16<br>AVX512VL | Compute the approximate reciprocals of the square roots of packed FP16 values in ymm2/m256/m16bcst and store the result in ymm1 subject to writemask k1. |
| EEX.512.66.MAP6.WO 4E /r<br>VRSQRTPH zmm1{k1}{z}, zmm2/<br>m512/m16bcst | A         | V/V                          | AVX512-FP16             | Compute the approximate reciprocals of the square roots of packed FP16 values in zmm2/m512/m16bcst and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction performs a SIMD computation of the approximate reciprocals square-root of 8/16/32 packed FP16 floating-point values in the source operand (the second operand) and stores the packed FP16 floating-point results in the destination operand.

The maximum relative error for this approximation is less than  $2^{-11} + 2^{-14}$ . For special cases, see Table 5-28.

The destination elements are updated according to the writemask.

**Table 5-28. VRSQRTPH/VRSQRTSH Special Cases**

| Input value   | Reset Value     | Comments                 |
|---------------|-----------------|--------------------------|
| Any denormal  | Normal          | Cannot generate overflow |
| $X = 2^{-2n}$ | $2^n$           |                          |
| $X < 0$       | QNaN_Indefinite | Including $-\infty$      |
| $X = -0$      | $-\infty$       |                          |
| $X = +0$      | $+\infty$       |                          |
| $X = +\infty$ | $+0$            |                          |



**Operation****VRSQRTPH dest{k1}, src**

VL = 128, 256 or 512

KL := VL/16

FOR i := 0 to KL-1:

IF k1[i] or \*no writemask\*:

IF SRC is memory and (EVEX.b = 1):

tsrc := src.fp16[0]

ELSE:

tsrc := src.fp16[i]

DEST.fp16[i] := APPROXIMATE(1.0 / SQRT(tsrc))

ELSE IF \*zeroing\*:

DEST.fp16[i] := 0

//else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VRSQRTPH \_\_m128h \_\_mm\_mask\_rsqrt\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a);

VRSQRTPH \_\_m128h \_\_mm\_maskz\_rsqrt\_ph (\_\_mmask8 k, \_\_m128h a);

VRSQRTPH \_\_m128h \_\_mm\_rsqrt\_ph (\_\_m128h a);

VRSQRTPH \_\_m256h \_\_mm256\_mask\_rsqrt\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a);

VRSQRTPH \_\_m256h \_\_mm256\_maskz\_rsqrt\_ph (\_\_mmask16 k, \_\_m256h a);

VRSQRTPH \_\_m256h \_\_mm256\_rsqrt\_ph (\_\_m256h a);

VRSQRTPH \_\_m512h \_\_mm512\_mask\_rsqrt\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a);

VRSQRTPH \_\_m512h \_\_mm512\_maskz\_rsqrt\_ph (\_\_mmask32 k, \_\_m512h a);

VRSQRTPH \_\_m512h \_\_mm512\_rsqrt\_ph (\_\_m512h a);

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-49, "Type E4 Class Exception Conditions."

## VRSQRTSH—Compute Approximate Reciprocal of Square Root of Scalar FP16 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|---|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.66.MAP6.WO 4F /r<br>VRSQRTSH xmm1{k1}{z}, xmm2,<br>xmm3/m16 | A         | V/V                          | AVX512-FP16           | Compute the approximate reciprocal square root of the FP16 value in xmm3/m16 and store the result in the low word element of xmm1 subject to writemask k1. Bits 127:16 of xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs the computation of the approximate reciprocal square-root of the low FP16 value in the second source operand (the third operand) and stores the result in the low word element of the destination operand (the first operand) according to the writemask k1.

The maximum relative error for this approximation is less than  $2^{-11} + 2^{-14}$ .

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed.

For special cases, see Table 5-28.

### Operation

**VRSQRTSH dest{k1}, src1, src2**

VL = 128, 256 or 512

KL := VL/16

IF k1[0] or \*no writemask\*:

DEST.fp16[0] := APPROXIMATE(1.0 / SQRT(src2.fp16[0]))

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]

DEST[MAXVL-1:128] := 0

### Intel C/C++ Compiler Intrinsic Equivalent

VRSQRTSH \_\_m128h \_\_mm\_mask\_rsqrt\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VRSQRTSH \_\_m128h \_\_mm\_maskz\_rsqrt\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VRSQRTSH \_\_m128h \_\_mm\_rsqrt\_sh (\_\_m128h a, \_\_m128h b);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

EVEX-encoded instruction, see Table 2-58, "Type E10 Class Exception Conditions."

## VSCALEFPD—Scale Packed Float64 Values With Float64 Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W1 2C /r<br>VSCALEFPD xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m64bcst     | A          | V/V                          | AVX512VL<br>AVX512F      | Scale the packed double precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1. |
| EVEX.256.66.0F38.W1 2C /r<br>VSCALEFPD ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m64bcst     | A          | V/V                          | AVX512VL<br>AVX512F      | Scale the packed double precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1. |
| EVEX.512.66.0F38.W1 2C /r<br>VSCALEFPD zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m64bcst{er} | A          | V/V                          | AVX512F                  | Scale the packed double precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Performs a floating-point scale of the packed double precision floating-point values in the first source operand by multiplying them by 2 to the power of the double precision floating-point values in second source operand.

The equation of this operation is given by:

$$\text{zmm1} := \text{zmm2} * 2^{\text{floor}(\text{zmm3})}$$

Floor(zmm3) means maximum integer value  $\leq$  zmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-29 and Table 5-30.

**Table 5-29. VSCALEFPD/SD/PS/SS Special Cases**

|      |             | Src2       |                       |                     |                | Set IE                   |
|------|-------------|------------|-----------------------|---------------------|----------------|--------------------------|
|      |             | $\pm$ NaN  | +Inf                  | -Inf                | 0/Denorm/Norm  |                          |
| Src1 | $\pm$ QNaN  | QNaN(Src1) | +INF                  | +0                  | QNaN(Src1)     | IF either source is SNAN |
|      | $\pm$ SNaN  | QNaN(Src1) | QNaN(Src1)            | QNaN(Src1)          | QNaN(Src1)     | YES                      |
|      | $\pm$ Inf   | QNaN(Src2) | Src1                  | QNaN_Indefinite     | Src1           | IF Src2 is SNAN or -INF  |
|      | $\pm$ 0     | QNaN(Src2) | QNaN_Indefinite       | Src1                | Src1           | IF Src2 is SNAN or +INF  |
|      | Denorm/Norm | QNaN(Src2) | $\pm$ INF (Src1 sign) | $\pm$ 0 (Src1 sign) | Compute Result | IF Src2 is SNAN          |

Table 5-30. Additional VSCALEFPD/SD Special Cases

| Special Case                    | Returned value  | Faults    |
|---------------------------------|---|-----------|
| $ \text{result}  < 2^{-1074}$   | $\pm 0$ or $\pm \text{Min-Denormal}$ (Src1 sign)                    | Underflow |
| $ \text{result}  \geq 2^{1024}$ | $\pm \text{INF}$ (Src1 sign) or $\pm \text{Max-normal}$ (Src1 sign) | Overflow  |

**Operation**

```

SCALE(SRC1, SRC2)
{
  TMP_SRC2 := SRC2
  TMP_SRC1 := SRC1
  IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
  IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
  /* SRC2 is a 64 bits floating-point value */
  DEST[63:0] := TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
VSCALEFPD (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
  THEN
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
  ELSE
    SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[63:0]);
      ELSE DEST[i+63:i] := SCALE(SRC1[i+63:i], SRC2[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] := 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFPD __m512d __mm512_scalef_round_pd(__m512d a, __m512d b, int rounding);
VSCALEFPD __m512d __mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d __mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int rounding);
VSCALEFPD __m512d __mm512_scalef_pd(__m512d a, __m512d b);
VSCALEFPD __m512d __mm512_mask_scalef_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m512d __mm512_maskz_scalef_pd(__mmask8 k, __m512d a, __m512d b);
VSCALEFPD __m256d __mm256_scalef_pd(__m256d a, __m256d b);
VSCALEFPD __m256d __mm256_mask_scalef_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m256d __mm256_maskz_scalef_pd(__mmask8 k, __m256d a, __m256d b);
VSCALEFPD __m128d __mm_scalef_pd(__m128d a, __m128d b);
VSCALEFPD __m128d __mm_mask_scalef_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VSCALEFPD __m128d __mm_maskz_scalef_pd(__mmask8 k, __m128d a, __m128d b);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).

Denormal is not reported for Src2.

**Other Exceptions**

See Table 2-46, “Type E2 Class Exception Conditions.”

## VSCALEFPH—Scale Packed FP16 Values with FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description   |
|---|-----------|------------------------------|-------------------------|---|
| EVEX.128.66.MAP6.W0 2C /r<br>VSCALEFPH xmm1{k1}{z}, xmm2,<br>xmm3/m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Scale the packed FP16 values in xmm2 using values from xmm3/m128/m16bcst, and store the result in xmm1 subject to writemask k1. |
| EVEX.256.66.MAP6.W0 2C /r<br>VSCALEFPH ymm1{k1}{z}, ymm2,<br>ymm3/m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Scale the packed FP16 values in ymm2 using values from ymm3/m256/m16bcst, and store the result in ymm1 subject to writemask k1. |
| EVEX.512.66.MAP6.W0 2C /r<br>VSCALEFPH zmm1{k1}{z}, zmm2,<br>zmm3/m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Scale the packed FP16 values in zmm2 using values from zmm3/m512/m16bcst, and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|---------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a floating-point scale of the packed FP16 values in the first source operand by multiplying it by 2 to the power of the FP16 values in second source operand. The destination elements are updated according to the writemask.

The equation of this operation is given by:

$$\text{zmm1} := \text{zmm2} * 2^{\text{floor}(\text{zmm3})}$$

Floor(zmm3) means maximum integer value  $\leq$  zmm3.

If the result cannot be represented in FP16, 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), and on the SAE bit.

Handling of special-case input values are listed in Table 5-31 and Table 5-32.

**Table 5-31. VSCALEFPH/VSCALEFSH Special Cases**

| Src1        | Src2       |                       |                     |                | Set IE                   |
|-------------|------------|-----------------------|---------------------|----------------|--------------------------|
|             | $\pm$ NaN  | +INF                  | -INF                | 0/Denorm/Norm  |                          |
| $\pm$ QNaN  | QNaN(Src1) | +INF                  | +0                  | QNaN(Src1)     | IF either source is SNaN |
| $\pm$ SNaN  | QNaN(Src1) | QNaN(Src1)            | QNaN(Src1)          | QNaN(Src1)     | YES                      |
| $\pm$ INF   | QNaN(Src2) | Src1                  | QNaN_Indefinite     | Src1           | IF Src2 is SNaN or -INF  |
| $\pm$ 0     | QNaN(Src2) | QNaN_Indefinite       | Src1                | Src1           | IF Src2 is SNaN or +INF  |
| Denorm/Norm | QNaN(Src2) | $\pm$ INF (Src1 sign) | $\pm$ 0 (Src1 sign) | Compute Result | IF Src2 is SNaN          |

**Table 5-32. Additional VSCALEFPH/VSCALEFSH Special Cases**

| Special Case                  | Returned Value  | Faults    |
|-------------------------------|---|-----------|
| $ \text{result}  < 2^{-24}$   | $\pm$ 0 or $\pm$ Min-Denormal (Src1 sign)               | Underflow |
| $ \text{result}  \geq 2^{16}$ | $\pm$ INF (Src1 sign) or $\pm$ Max-Denormal (Src1 sign) | Overflow  |

**Operation**

```
def scale_fp16(src1,src2):
    tmp1 := src1
    tmp2 := src2
    return tmp1 * POW(2, FLOOR(tmp2))
```

**VSCALEFPH dest{k1}, src1, src2**

VL = 128, 256, or 512

KL := VL / 16

IF (VL = 512) AND (EVEX.b = 1) and no memory operand:

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR i := 0 to KL-1:

IF k1[i] or \*no writemask\*:

IF SRC2 is memory and (EVEX.b = 1):

tsrc := src2.fp16[0]

ELSE:

tsrc := src2.fp16[i]

dest.fp16[i] := scale\_fp16(src1.fp16[i],tsrc)

ELSE IF \*zeroing\*:

dest.fp16[i] := 0

//else dest.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VSCALEFPH \_\_m128h \_\_mm\_mask\_scalef\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VSCALEFPH \_\_m128h \_\_mm\_maskz\_scalef\_ph (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VSCALEFPH \_\_m128h \_\_mm\_scalef\_ph (\_\_m128h a, \_\_m128h b);

VSCALEFPH \_\_m256h \_\_mm256\_mask\_scalef\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a, \_\_m256h b);

VSCALEFPH \_\_m256h \_\_mm256\_maskz\_scalef\_ph (\_\_mmask16 k, \_\_m256h a, \_\_m256h b);

VSCALEFPH \_\_m256h \_\_mm256\_scalef\_ph (\_\_m256h a, \_\_m256h b);

VSCALEFPH \_\_m512h \_\_mm512\_mask\_scalef\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, \_\_m512h b);

VSCALEFPH \_\_m512h \_\_mm512\_maskz\_scalef\_ph (\_\_mmask32 k, \_\_m512h a, \_\_m512h b);

VSCALEFPH \_\_m512h \_\_mm512\_scalef\_ph (\_\_m512h a, \_\_m512h b);

VSCALEFPH \_\_m512h \_\_mm512\_mask\_scalef\_round\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, \_\_m512h b, const int rounding);

VSCALEFPH \_\_m512h \_\_mm512\_maskz\_scalef\_round\_ph (\_\_mmask32 k, \_\_m512h a, \_\_m512h b, const int);

VSCALEFPH \_\_m512h \_\_mm512\_scalef\_round\_ph (\_\_m512h a, \_\_m512h b, const int rounding);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-46, "Type E2 Class Exception Conditions".

Denormal-operand exception (#D) is checked and signaled for src1 operand, but not for src2 operand. The denormal-operand exception is checked for src1 operand only if the src2 operand is not NaN. If the src2 operand is NaN, the processor generates NaN and does not signal denormal-operand exception, even if src1 operand is denormal.

## VSCALEFPS—Scale Packed Float32 Values With Float32 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|-----------|------------------------------|--------------------------|---|
| EVEX.128.66.0F38.W0 2C /r<br>VSCALEFPS xmm1 {k1}{z}, xmm2,<br>xmm3/m128/m32bcst     | A         | V/V                          | AVX512VL<br>AVX512F      | Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.                |
| EVEX.256.66.0F38.W0 2C /r<br>VSCALEFPS ymm1 {k1}{z}, ymm2,<br>ymm3/m256/m32bcst     | A         | V/V                          | AVX512VL<br>AVX512F      | Scale the packed single-precision values in ymm2 using floating-point values from ymm3/m256/m32bcst. Under writemask k1.                |
| EVEX.512.66.0F38.W0 2C /r<br>VSCALEFPS zmm1 {k1}{z}, zmm2,<br>zmm3/m512/m32bcst{er} | A         | V/V                          | AVX512F                  | Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying them by 2 to the power of the float32 values in second source operand.

The equation of this operation is given by:

$$\text{zmm1} := \text{zmm2} * 2^{\text{floor}(\text{zmm3})}$$

Floor(zmm3) means maximum integer value  $\leq$  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 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.

Handling of special-case input values are listed in Table 5-29 and Table 5-33.

**Table 5-33. Additional VSCALEFPS/SS Special Cases**

| Special Case                   | Returned value  | Faults    |
|--------------------------------|---|-----------|
| $ \text{result}  < 2^{-149}$   | $\pm 0$ or $\pm \text{Min-Denormal}$ (Src1 sign)                    | Underflow |
| $ \text{result}  \geq 2^{128}$ | $\pm \text{INF}$ (Src1 sign) or $\pm \text{Max-normal}$ (Src1 sign) | Overflow  |



**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 __mm_scalef_ps(__m128 a, __m128 b);
VSCALEFPS __m128 __mm_mask_scalef_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VSCALEFPS __m128 __mm_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.”

## VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

| Opcode/<br>Instruction  | Op /<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|------------|------------------------------|--------------------------|--|
| EVEX.LLIG.66.0F38.W1 2D /r<br>VSCALEFSD xmm1 {k1}{z}, xmm2,<br>xmm3/m64{er} | A          | V/V                          | AVX512F                  | Scale the scalar double precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Performs a floating-point scale of the scalar double precision floating-point value in the first source operand by multiplying it by 2 to the power of the double precision floating-point value in second source operand.

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{floor}(\text{xmm3})}$$

Floor(xmm3) means maximum integer value  $\leq$  xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-29 and Table 5-30.

### Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 := SRC2
    TMP_SRC1 := SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 64 bits floating-point value */
    DEST[63:0] := TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

**VSCALEFSD (EVEX encoded version)**

```

IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(EVEX.RC);
    ELSE
        SET_ROUNDING_MODE_FOR_THIS_INSTRUCTION(MXCSR.RC);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[63:0] := SCALE(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[63:0] := 0
        FI
FI;
DEST[127:64] := SRC1[127:64]
DEST[MAXVL-1:128] := 0

```

**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSCALEFSD __m128d __mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d __mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d __mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).

Denormal is not reported for Src2.

**Other Exceptions**

See Table 2-47, "Type E3 Class Exception Conditions."

## VSCALEFSH—Scale Scalar FP16 Values with FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|---|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.66.MAP6.WO 2D /r<br>VSCALEFSH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Scale the FP16 values in xmm2 using the value from xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a floating-point scale of the low FP16 element in the first source operand by multiplying it by 2 to the power of the low FP16 element in second source operand, storing the result in the low element of the destination operand.

Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{floor}(\text{xmm3})}$$

Floor(xmm3) means maximum integer value  $\leq$  xmm3.

If the result cannot be represented in FP16, 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.

Handling of special-case input values are listed in Table 5-31 and Table 5-32.

### Operation

#### VSCALEFSH dest{k1}, src1, src2

IF (EVEX.b = 1) and no memory operand:

```
SET_RM(EVEX.RC)
```

ELSE

```
SET_RM(MXCSR.RC)
```

IF k1[0] or \*no writemask\*:

```
dest.fp16[0] := scale_fp16(src1.fp16[0], src2.fp16[0]) // see VSCALEFPH
```

ELSE IF \*zeroing\*:

```
dest.fp16[0] := 0
```

//else DEST.fp16[0] remains unchanged

```
DEST[127:16] := src1[127:16]
```

```
DEST[MAXVL-1:128] := 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSH \_\_m128h \_\_mm\_mask\_scalef\_round\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b, const int rounding);  
VSCALEFSH \_\_m128h \_\_mm\_maskz\_scalef\_round\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, const int rounding);  
VSCALEFSH \_\_m128h \_\_mm\_scalef\_round\_sh (\_\_m128h a, \_\_m128h b, const int rounding);  
VSCALEFSH \_\_m128h \_\_mm\_mask\_scalef\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
VSCALEFSH \_\_m128h \_\_mm\_maskz\_scalef\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);  
VSCALEFSH \_\_m128h \_\_mm\_scalef\_sh (\_\_m128h a, \_\_m128h b);

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, “Type E3 Class Exception Conditions.”

Denormal-operand exception (#D) is checked and signaled for src1 operand, but not for src2 operand. The denormal-operand exception is checked for src1 operand only if the src2 operand is not NaN. If the src2 operand is NaN, the processor generates NaN and does not signal denormal-operand exception, even if src1 operand is denormal.

## VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---|-----------|------------------------------|--------------------------|--|
| EVEX.LLIG.66.0F38.W0 2D /r<br>VSCALEFSS xmm1 {k1}{z}, xmm2,<br>xmm3/m32{er} | A         | V/V                          | AVX512F                  | Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|---------------|---------------|---------------|---------------|-----------|
| A     | Tuple1 Scalar | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 to the power of the float32 value in second source operand.

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{floor}(\text{xmm3})}$$

Floor(xmm3) means maximum integer value  $\leq$  xmm3.

If the result cannot be represented in single-precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-29 and Table 5-33.

**Operation**

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 := SRC2
    TMP_SRC1 := SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] := TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

**VSCALEFSS (EVEX encoded version)**

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_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[127:32] := SRC1[127:32]
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

| Opcode/<br>Instruction                                       | Op/En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|--|-------|------------------------------|--------------------------|--|
| EVEX.128.66.0F38.W0 A2 /vsib<br>VSCATTERDPS vm32x {k1}, xmm1 | A     | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.256.66.0F38.W0 A2 /vsib<br>VSCATTERDPS vm32y {k1}, ymm1 | A     | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.512.66.0F38.W0 A2 /vsib<br>VSCATTERDPS vm32z {k1}, zmm1 | A     | V/V                          | AVX512F                  | Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.128.66.0F38.W1 A2 /vsib<br>VSCATTERDPD vm32x {k1}, xmm1 | A     | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, scatter double precision floating-point values to memory using writemask k1. |
| EVEX.256.66.0F38.W1 A2 /vsib<br>VSCATTERDPD vm32y {k1}, ymm1 | A     | V/V                          | AVX512VL<br>AVX512F      | Using signed dword indices, scatter double precision floating-point values to memory using writemask k1. |
| EVEX.512.66.0F38.W1 A2 /vsib<br>VSCATTERDPD vm32y {k1}, zmm1 | A     | V/V                          | AVX512F                  | Using signed dword indices, scatter double precision floating-point values to memory using writemask k1. |
| EVEX.128.66.0F38.W0 A3 /vsib<br>VSCATTERQPS vm64x {k1}, xmm1 | A     | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.256.66.0F38.W0 A3 /vsib<br>VSCATTERQPS vm64y {k1}, xmm1 | A     | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.512.66.0F38.W0 A3 /vsib<br>VSCATTERQPS vm64z {k1}, ymm1 | A     | V/V                          | AVX512F                  | Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1. |
| EVEX.128.66.0F38.W1 A3 /vsib<br>VSCATTERQPD vm64x {k1}, xmm1 | A     | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, scatter double precision floating-point values to memory using writemask k1. |
| EVEX.256.66.0F38.W1 A3 /vsib<br>VSCATTERQPD vm64y {k1}, ymm1 | A     | V/V                          | AVX512VL<br>AVX512F      | Using signed qword indices, scatter double precision floating-point values to memory using writemask k1. |
| EVEX.512.66.0F38.W1 A3 /vsib<br>VSCATTERQPD vm64z {k1}, zmm1 | A     | V/V                          | AVX512F                  | Using signed qword indices, scatter double precision floating-point values to memory using writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple Type    | Operand 1   | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---|---------------|-----------|-----------|
| A     | Tuple1 Scalar | BaseReg (R): VSIB:base,<br>VectorReg(R): VSIB:index | ModRM:reg (r) | N/A       | N/A       |

### 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  $\text{disp8} * N$  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 + (VINDEK[k+63:k]) * SCALE + DISP] :=
            SRC[j+31:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0

```

```

VSCATTERQPD (EVEX encoded versions)
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j := 0 TO KL-1
    i := j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEK[i+63:i]) * SCALE + DISP] :=
            SRC[j+63:i]
            k1[j] := 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] := 0

```

#### Intel C/C++ Compiler Intrinsic Equivalent

```

VSCATTERDPD void __mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERDPD void __mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void __mm512_i32scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERDPS void __mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPD void __mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void __mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void __mm512_i64scatter_ps(void * base, __m512i vdx, __m256 a, int scale);
VSCATTERQPS void __mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m256 a, int scale);
VSCATTERDPD void __mm256_i32scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERDPD void __mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m256d a, int scale);
VSCATTERDPS void __mm256_i32scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERDPS void __mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void __mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void __mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void __mm256_i64scatter_ps(void * base, __m256i vdx, __m128 a, int scale);
VSCATTERQPS void __mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m128 a, int scale);
VSCATTERDPD void __mm_i32scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERDPD void __mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERDPS void __mm_i32scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERDPS void __mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERQPD void __mm_i64scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERQPD void __mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERQPS void __mm_i64scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERQPS void __mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 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/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-Bit Granularity

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|--|-----------|------------------------------|--------------------------|---|
| EVEX.256.66.0F3A.W0 23 /r ib<br>VSHUFF32X4 ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F3A.W0 23 /r ib<br>VSHUFF32x4 zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8 | A         | V/V                          | AVX512F                  | 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. |
| EVEX.256.66.0F3A.W1 23 /r ib<br>VSHUFF64X2 ymm1{k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512F      | 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. |
| EVEX.512.66.0F3A.W1 23 /r ib<br>VSHUFF64x2 zmm1{k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | A         | V/V                          | AVX512F                  | 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. |
| EVEX.256.66.0F3A.W0 43 /r ib<br>VSHUFI32X4 ymm1{k1}{z}, ymm2,<br>ymm3/m256/m32bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512F      | 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.                     |
| EVEX.512.66.0F3A.W0 43 /r ib<br>VSHUFI32x4 zmm1{k1}{z}, zmm2,<br>zmm3/m512/m32bcst, imm8 | A         | V/V                          | AVX512F                  | 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.                     |
| EVEX.256.66.0F3A.W1 43 /r ib<br>VSHUFI64X2 ymm1{k1}{z}, ymm2,<br>ymm3/m256/m64bcst, imm8 | A         | V/V                          | AVX512VL<br>AVX512F      | 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.                       |
| EVEX.512.66.0F3A.W1 43 /r ib<br>VSHUFI64x2 zmm1{k1}{z}, zmm2,<br>zmm3/m512/m64bcst, imm8 | A         | V/V                          | AVX512F                  | 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.                       |

### Instruction Operand Encoding

| Op/En | Tuple Type | Operand 1     | Operand 2     | Operand 3     | Operand 4 |
|-------|------------|---------------|---------------|---------------|-----------|
| A     | Full       | ModRM:reg (w) | EVEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

256-bit Version: Moves one of the two 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

512-bit Version: Moves two of the four 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.

### Operation

```
Select2(SRC, control) {
CASE (control[0]) OF
  0:  TMP := SRC[127:0];
  1:  TMP := SRC[255:128];
ESAC;
RETURN TMP
}
```

```
Select4(SRC, control) {
CASE (control[1:0]) OF
  0:  TMP := SRC[127:0];
  1:  TMP := SRC[255:128];
  2:  TMP := SRC[383:256];
  3:  TMP := SRC[511:384];
ESAC;
RETURN TMP
}
```

### VSHUFF32x4 (EVEX versions)

(KL, VL) = (8, 256), (16, 512)

```
FOR j := 0 TO KL-1
  i := j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] := SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
  i := j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] := TMP_DEST[i+31:i]
    ELSE
      IF *merging-masking* ; 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[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
  i := j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] := TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          THEN DEST[i+63:i] := 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL] := 0

```

**VSHUFI32x4 (EVEX 512-bit version)**

(KL, VL) = (8, 256), (16, 512)

```

FOR j := 0 TO KL-1
  i := j * 32
  IF (EVEX.b = 1) AND (SRC2 *is memory*)
    THEN TMP_SRC2[i+31:i] := SRC2[31:0]
    ELSE TMP_SRC2[i+31:i] := SRC2[i+31:i]
  FI;
ENDFOR;
IF VL = 256
  TMP_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);
  TMP_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
  TMP_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);
  TMP_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);
  TMP_DEST[383:256] := Select4(TMP_SRC2[511:0], imm8[5:4]);
  TMP_DEST[511:384] := Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j := 0 TO KL-1
  i := j * 32

```

```

IF k1[j] OR *no writemask*
  THEN DEST[i+31:i] := TMP_DEST[i+31:i]
  ELSE
    IF *merging-masking*           ; 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

```

**VSHUFI64x2 (EVEX 512-bit version)**

(KL, VL) = (4, 256), (8, 512)

FOR j := 0 TO KL-1

i := j \* 64

IF (EVEX.b = 1) AND (SRC2 \*is memory\*)

THEN TMP\_SRC2[i+63:i] := SRC2[63:0]

ELSE TMP\_SRC2[i+63:i] := SRC2[i+63:i]

FI;

ENDFOR;

IF VL = 256

TMP\_DEST[127:0] := Select2(SRC1[255:0], imm8[0]);

TMP\_DEST[255:128] := Select2(SRC2[255:0], imm8[1]);

FI;

IF VL = 512

TMP\_DEST[127:0] := Select4(SRC1[511:0], imm8[1:0]);

TMP\_DEST[255:128] := Select4(SRC1[511:0], imm8[3:2]);

TMP\_DEST[383:256] := Select4(TMP\_SRC2[511:0], imm8[5:4]);

TMP\_DEST[511:384] := Select4(TMP\_SRC2[511:0], imm8[7:6]);

FI;

FOR j := 0 TO KL-1

i := j \* 64

IF k1[j] OR \*no writemask\*

THEN DEST[i+63:i] := TMP\_DEST[i+63:i]

ELSE

IF \*merging-masking\* ; merging-masking

THEN \*DEST[i+63:i] remains unchanged\*

ELSE \*zeroing-masking\* ; zeroing-masking

THEN DEST[i+63:i] := 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] := 0



**Intel C/C++ Compiler Intrinsic Equivalent**

```

VSHUFI32x4 __m512i __mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i __mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i __mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i __mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i __mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i __mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m512 __mm512_shuffle_f32x4(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 __mm512_mask_shuffle_f32x4(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 __mm512_maskz_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFI64x2 __m512i __mm512_shuffle_i64x2(__m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i __mm512_mask_shuffle_i64x2(__m512i s, __mmask8 k, __m512i b, __m512i b, int imm);
VSHUFI64x2 __m512i __mm512_maskz_shuffle_i64x2(__mmask8 k, __m512i a, __m512i b, int imm);
VSHUFF64x2 __m512d __mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d __mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d __mm512_maskz_shuffle_f64x2(__mmask8 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.

## VSQRTPH—Compute Square Root of Packed FP16 Values

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|--|-----------|------------------------------|-------------------------|--|
| EVEX.128.NP.MAP5.W0 51 /r<br>VSQRTPH xmm1{k1}{z}, xmm2/m128/<br>m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Compute square roots of the packed FP16 values in xmm2/m128/m16bcst, and store the result in xmm1 subject to writemask k1. |
| EVEX.256.NP.MAP5.W0 51 /r<br>VSQRTPH ymm1{k1}{z}, ymm2/m256/<br>m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Compute square roots of the packed FP16 values in ymm2/m256/m16bcst, and store the result in ymm1 subject to writemask k1. |
| EVEX.512.NP.MAP5.W0 51 /r<br>VSQRTPH zmm1{k1}{z}, zmm2/m512/<br>m16bcst {er} | A         | V/V                          | AVX512-FP16             | Compute square roots of the packed FP16 values in zmm2/m512/m16bcst, and store the result in zmm1 subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|-------|---------------|---------------|-----------|-----------|
| A     | Full  | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction performs a packed FP16 square-root computation on the values from source operand and stores the packed FP16 result in the destination operand. The destination elements are updated according to the write-mask.

### Operation

**VSQRTPH dest{k1}, src**

VL = 128, 256 or 512

KL := VL/16

FOR i := 0 to KL-1:

  IF k1[i] or \*no writemask\*:

    IF SRC is memory and (EVEX.b = 1):

      tsrc := src.fp16[0]

    ELSE:

      tsrc := src.fp16[i]

      DEST.fp16[i] := SQRT(tsrc)

  ELSE IF \*zeroing\*:

    DEST.fp16[i] := 0

  //else DEST.fp16[i] remains unchanged

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VSQRTPH \_\_m128h \_mm\_mask\_sqrt\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a);  
 VSQRTPH \_\_m128h \_mm\_maskz\_sqrt\_ph (\_\_mmask8 k, \_\_m128h a);  
 VSQRTPH \_\_m128h \_mm\_sqrt\_ph (\_\_m128h a);  
 VSQRTPH \_\_m256h \_mm256\_mask\_sqrt\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a);  
 VSQRTPH \_\_m256h \_mm256\_maskz\_sqrt\_ph (\_\_mmask16 k, \_\_m256h a);  
 VSQRTPH \_\_m256h \_mm256\_sqrt\_ph (\_\_m256h a);  
 VSQRTPH \_\_m512h \_mm512\_mask\_sqrt\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a);  
 VSQRTPH \_\_m512h \_mm512\_maskz\_sqrt\_ph (\_\_mmask32 k, \_\_m512h a);  
 VSQRTPH \_\_m512h \_mm512\_sqrt\_ph (\_\_m512h a);  
 VSQRTPH \_\_m512h \_mm512\_mask\_sqrt\_round\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, const int rounding);  
 VSQRTPH \_\_m512h \_mm512\_maskz\_sqrt\_round\_ph (\_\_mmask32 k, \_\_m512h a, const int rounding);  
 VSQRTPH \_\_m512h \_mm512\_sqrt\_round\_ph (\_\_m512h a, const int rounding);

**SIMD Floating-Point Exceptions**

Invalid, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-46, “Type E2 Class Exception Conditions.”

## VSQRTSH—Compute Square Root of Scalar FP16 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEEX.LLIG.F3.MAP5.W0 51 /r<br>VSQRTSH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Compute square root of the low FP16 value in xmm3/m16 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction performs a scalar FP16 square-root computation on the source operand and stores the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

**VSQRTSH dest{k1}, src1, src2**

IF k1[0] or \*no writemask\*:

DEST.fp16[0] := Sqrt(src2.fp16[0])

ELSE IF \*zeroing\*:

DEST.fp16[0] := 0

//else DEST.fp16[0] remains unchanged

DEST[127:16] := src1[127:16]

DEST[MAXVL-1:128] := 0

### Intel C/C++ Compiler Intrinsic Equivalent

VSQRTSH \_\_m128h \_\_mm\_mask\_sqrt\_round\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b, const int rounding);

VSQRTSH \_\_m128h \_\_mm\_maskz\_sqrt\_round\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b, const int rounding);

VSQRTSH \_\_m128h \_\_mm\_sqrt\_round\_sh (\_\_m128h a, \_\_m128h b, const int rounding);

VSQRTSH \_\_m128h \_\_mm\_mask\_sqrt\_sh (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VSQRTSH \_\_m128h \_\_mm\_maskz\_sqrt\_sh (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VSQRTSH \_\_m128h \_\_mm\_sqrt\_sh (\_\_m128h a, \_\_m128h b);

### SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

### Other Exceptions

EVEEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

## VSUBPH—Subtract Packed FP16 Values

| Opcode/<br>Instruction  | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag   | Description  |
|---|-----------|------------------------------|-------------------------|--|
| EVEX.128.NP.MAP5.W0 5C /r<br>VSUBPH xmm1{k1}{z}, xmm2, xmm3/<br>m128/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Subtract packed FP16 values from xmm3/m128/<br>m16bcst to xmm2, and store the result in xmm1<br>subject to writemask k1. |
| EVEX.256.NP.MAP5.W0 5C /r<br>VSUBPH ymm1{k1}{z}, ymm2, ymm3/<br>m256/m16bcst      | A         | V/V                          | AVX512-FP16<br>AVX512VL | Subtract packed FP16 values from ymm3/m256/<br>m16bcst to ymm2, and store the result in ymm1<br>subject to writemask k1. |
| EVEX.512.NP.MAP5.W0 5C /r<br>VSUBPH zmm1{k1}{z}, zmm2, zmm3/<br>m512/m16bcst {er} | A         | V/V                          | AVX512-FP16             | Subtract packed FP16 values from zmm3/m512/<br>m16bcst to zmm2, and store the result in zmm1<br>subject to writemask k1. |

### Instruction Operand Encoding

| Op/En | Tuple | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|-------|---------------|--------------|---------------|-----------|
| A     | Full  | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction subtracts packed FP16 values from second source operand from the corresponding elements in the first source operand, storing the packed FP16 result in the destination operand. The destination elements are updated according to the writemask.

### Operation

**VSUBPH (EVEX encoded versions) when src2 operand is a register**

VL = 128, 256 or 512

KL := VL/16

IF (VL = 512) AND (EVEX.b = 1):

SET\_RM(EVEX.RC)

ELSE

SET\_RM(MXCSR.RC)

FOR j := 0 TO KL-1:

IF k1[j] OR \*no writemask\*:

DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**VSUBPH (EVEX encoded versions) when src2 operand is a memory source**

VL = 128, 256 or 512

KL := VL/16

FOR j := 0 TO KL-1:

IF k1[j] OR \*no writemask\*:

IF EVEX.b = 1:

DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[0]

ELSE:

DEST.fp16[j] := SRC1.fp16[j] - SRC2.fp16[j]

ELSE IF \*zeroing\*:

DEST.fp16[j] := 0

// else dest.fp16[j] remains unchanged

DEST[MAXVL-1:VL] := 0

**Intel C/C++ Compiler Intrinsic Equivalent**

VSUBPH \_\_m128h \_\_mm\_mask\_sub\_ph (\_\_m128h src, \_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VSUBPH \_\_m128h \_\_mm\_maskz\_sub\_ph (\_\_mmask8 k, \_\_m128h a, \_\_m128h b);

VSUBPH \_\_m128h \_\_mm\_sub\_ph (\_\_m128h a, \_\_m128h b);

VSUBPH \_\_m256h \_\_mm256\_mask\_sub\_ph (\_\_m256h src, \_\_mmask16 k, \_\_m256h a, \_\_m256h b);

VSUBPH \_\_m256h \_\_mm256\_maskz\_sub\_ph (\_\_mmask16 k, \_\_m256h a, \_\_m256h b);

VSUBPH \_\_m256h \_\_mm256\_sub\_ph (\_\_m256h a, \_\_m256h b);

VSUBPH \_\_m512h \_\_mm512\_mask\_sub\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, \_\_m512h b);

VSUBPH \_\_m512h \_\_mm512\_maskz\_sub\_ph (\_\_mmask32 k, \_\_m512h a, \_\_m512h b);

VSUBPH \_\_m512h \_\_mm512\_sub\_ph (\_\_m512h a, \_\_m512h b);

VSUBPH \_\_m512h \_\_mm512\_mask\_sub\_round\_ph (\_\_m512h src, \_\_mmask32 k, \_\_m512h a, \_\_m512h b, int rounding);

VSUBPH \_\_m512h \_\_mm512\_maskz\_sub\_round\_ph (\_\_mmask32 k, \_\_m512h a, \_\_m512h b, int rounding);

VSUBPH \_\_m512h \_\_mm512\_sub\_round\_ph (\_\_m512h a, \_\_m512h b, int rounding);

**SIMD Floating-Point Exceptions**

Invalid, Underflow, Overflow, Precision, Denormal.

**Other Exceptions**

EVEX-encoded instruction, see Table 2-46, "Type E2 Class Exception Conditions."

## VSUBSH—Subtract Scalar FP16 Value

| Opcode/<br>Instruction   | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description   |
|--|-----------|------------------------------|-----------------------|---|
| EVEX.LLIG.F3.MAP5.WO 5C /r<br>VSUBSH xmm1{k1}{z}, xmm2,<br>xmm3/m16 {er} | A         | V/V                          | AVX512-FP16           | Subtract the low FP16 value in xmm3/m16 from xmm2 and store the result in xmm1 subject to writemask k1. Bits 127:16 from xmm2 are copied to xmm1[127:16]. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2    | Operand 3     | Operand 4 |
|-------|--------|---------------|--------------|---------------|-----------|
| A     | Scalar | ModRM:reg (w) | VEX.vvvv (r) | ModRM:r/m (r) | N/A       |

### Description

This instruction subtracts the low FP16 value from the second source operand from the corresponding value in the first source operand, storing the FP16 result in the destination operand. Bits 127:16 of the destination operand are copied from the corresponding bits of the first source operand. Bits MAXVL-1:128 of the destination operand are zeroed. The low FP16 element of the destination is updated according to the writemask.

### Operation

#### VSUBSH (EVEX encoded versions)

IF EVEX.b = 1 and SRC2 is a register:

```
SET_RM(EVEX.RC)
```

ELSE

```
SET_RM(MXCSR.RC)
```

IF k1[0] OR \*no writemask\*:

```
DEST.fp16[0] := SRC1.fp16[0] - SRC2.fp16[0]
```

ELSE IF \*zeroing\*:

```
DEST.fp16[0] := 0
```

// else dest.fp16[0] remains unchanged

```
DEST[127:16] := SRC1[127:16]
```

```
DEST[MAXVL-1:128] := 0
```

### Intel C/C++ Compiler Intrinsic Equivalent

```
VSUBSH __m128h __mm_mask_sub_round_sh (__m128h src, __mmask8 k, __m128h a, __m128h b, int rounding);
```

```
VSUBSH __m128h __mm_maskz_sub_round_sh (__mmask8 k, __m128h a, __m128h b, int rounding);
```

```
VSUBSH __m128h __mm_sub_round_sh (__m128h a, __m128h b, int rounding);
```

```
VSUBSH __m128h __mm_mask_sub_sh (__m128h src, __mmask8 k, __m128h a, __m128h b);
```

```
VSUBSH __m128h __mm_maskz_sub_sh (__mmask8 k, __m128h a, __m128h b);
```

```
VSUBSH __m128h __mm_sub_sh (__m128h a, __m128h b);
```

### SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal.

### Other Exceptions

EVEX-encoded instructions, see Table 2-47, "Type E3 Class Exception Conditions."

## VTESTPD/VTESTPS—Packed Bit Test

| Opcode/<br>Instruction                              | Op<br>/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description   |
|---|---------------|------------------------------|--------------------------|---|
| VEX.128.66.0F38.W0 0E /r<br>VTESTPS xmm1, xmm2/m128 | RM            | V/V                          | AVX                      | Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources. |
| VEX.256.66.0F38.W0 0E /r<br>VTESTPS ymm1, ymm2/m256 | RM            | V/V                          | AVX                      | Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources. |
| VEX.128.66.0F38.W0 0F /r<br>VTESTPD xmm1, xmm2/m128 | RM            | V/V                          | AVX                      | Set ZF and CF depending on sign bit AND and ANDN of packed double precision floating-point sources. |
| VEX.256.66.0F38.W0 0F /r<br>VTESTPD ymm1, ymm2/m256 | RM            | V/V                          | AVX                      | Set ZF and CF depending on sign bit AND and ANDN of packed double precision floating-point sources. |

### Instruction Operand Encoding

| Op/En | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|---------------|---------------|-----------|-----------|
| RM    | ModRM:reg (r) | ModRM:r/m (r) | N/A       | N/A       |

### 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)

```
TEMP[127:0] := SRC[127:0] AND DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
```

```
TEMP[127:0] := SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN CF := 1;
    ELSE CF := 0;
```

```
DEST (unmodified)
AF := OF := PF := SF := 0;
```

### VTESTPS (VEX.256 encoded version)

```
TEMP[255:0] := SRC[255:0] AND DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = TEMP[160] = TEMP[191] = TEMP[224] = TEMP[255] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
```

```
TEMP[255:0] := SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = TEMP[160] = TEMP[191] = TEMP[224] = TEMP[255] = 0)
    THEN CF := 1;
    ELSE CF := 0;
```

```
DEST (unmodified)
AF := OF := PF := SF := 0;
```

### VTESTPD (128-bit version)

```
TEMP[127:0] := SRC[127:0] AND DEST[127:0]
IF (TEMP[63] = TEMP[127] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
```

```
TEMP[127:0] := SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[63] = TEMP[127] = 0)
    THEN CF := 1;
    ELSE CF := 0;
```

```
DEST (unmodified)
AF := OF := PF := SF := 0;
```

### VTESTPD (VEX.256 encoded version)

```
TEMP[255:0] := SRC[255:0] AND DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN ZF := 1;
    ELSE ZF := 0;
```

```
TEMP[255:0] := SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN CF := 1;
    ELSE CF := 0;
```

```
DEST (unmodified)
AF := OF := PF := SF := 0;
```

**Intel C/C++ Compiler Intrinsic Equivalent****VTESTPS**

```
int _mm256_testz_ps (__m256 s1, __m256 s2);
int _mm256_testc_ps (__m256 s1, __m256 s2);
int _mm256_testnzc_ps (__m256 s1, __m128 s2);
int _mm_testz_ps (__m128 s1, __m128 s2);
int _mm_testc_ps (__m128 s1, __m128 s2);
int _mm_testnzc_ps (__m128 s1, __m128 s2);
```

**VTESTPD**

```
int _mm256_testz_pd (__m256d s1, __m256d s2);
int _mm256_testc_pd (__m256d s1, __m256d s2);
int _mm256_testnzc_pd (__m256d s1, __m256d s2);
int _mm_testz_pd (__m128d s1, __m128d s2);
int _mm_testc_pd (__m128d s1, __m128d s2);
int _mm_testnzc_pd (__m128d s1, __m128d s2);
```

**Flags Affected**

The OF, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Table 2-21, “Type 4 Class Exception Conditions.”

Additionally:

|     |                                      |
|-----|--------------------------------------|
| #UD | If VEX.vvvv ≠ 1111B.                 |
|     | If VEX.W = 1 for VTESTPS or VTESTPD. |

## VUCOMISH—Unordered Compare Scalar FP16 Values and Set EFLAGS

| Opcode/<br>Instruction                                      | Op/<br>En | 64/32<br>bit Mode<br>Support | CPUID Feature<br>Flag | Description  |
|---|-----------|------------------------------|-----------------------|--|
| EVEX.LLIG.NP.MAP5.WO 2E /r<br>VUCOMISH xmm1, xmm2/m16 {sae} | A         | V/V                          | AVX512-FP16           | Compare low FP16 values in xmm1 and xmm2/<br>m16 and set the EFLAGS flags accordingly. |

### Instruction Operand Encoding

| Op/En | Tuple  | Operand 1     | Operand 2     | Operand 3 | Operand 4 |
|-------|--------|---------------|---------------|-----------|-----------|
| A     | Scalar | ModRM:reg (w) | ModRM:r/m (r) | N/A       | N/A       |

### Description

This instruction compares the FP16 values in the low word of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 16-bit memory location.

The VUCOMISH instruction differs from the VCOMISH instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated. EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

### Operation

#### VUCOMISH

RESULT := UnorderedCompare(SRC1.fp16[0],SRC2.fp16[0])

if RESULT is UNORDERED:

ZF, PF, CF := 1, 1, 1

else if RESULT is GREATER\_THAN:

ZF, PF, CF := 0, 0, 0

else if RESULT is LESS\_THAN:

ZF, PF, CF := 0, 0, 1

else // RESULT is EQUALS

ZF, PF, CF := 1, 0, 0

OF, AF, SF := 0, 0, 0

### Intel C/C++ Compiler Intrinsic Equivalent

```
VUCOMISH int __mm_ucomieq_sh (__m128h a, __m128h b);
VUCOMISH int __mm_ucomige_sh (__m128h a, __m128h b);
VUCOMISH int __mm_ucomigt_sh (__m128h a, __m128h b);
VUCOMISH int __mm_ucomile_sh (__m128h a, __m128h b);
VUCOMISH int __mm_ucomilt_sh (__m128h a, __m128h b);
VUCOMISH int __mm_ucomineq_sh (__m128h a, __m128h b);
```

### SIMD Floating-Point Exceptions

Invalid, Denormal.

### Other Exceptions

EVEX-encoded instructions, see Table 2-48, “Type E3NF Class Exception Conditions.”

## VZEROALL—Zero XMM, YMM, and ZMM Registers

| Opcode/<br>Instruction        | Op<br>/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description                                   |
|-------------------------------|---------------|------------------------------|--------------------------|---|
| VEX.256.OF.WIG 77<br>VZEROALL | Z0            | V/V                          | AVX                      | Zero some of the XMM, YMM, and ZMM registers. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|-------|-----------|-----------|-----------|-----------|
| Z0    | N/A       | N/A       | N/A       | N/A       |

### Description

In 64-bit mode, the instruction zeroes XMM0-XMM15, YMM0-YMM15, and ZMM0-ZMM15. Outside 64-bit mode, it zeroes only XMM0-XMM7, YMM0-YMM7, and ZMM0-ZMM7. VZEROALL does not modify ZMM16-ZMM31.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

### Operation

`simd_reg_file[][]` is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm, and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0, and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512.

### VZEROALL (VEX.256 encoded version)

IF (64-bit mode)

  limit := 15

ELSE

  limit := 7

FOR i in 0 .. limit:

`simd_reg_file[i][MAXVL-1:0] := 0`

### Intel C/C++ Compiler Intrinsic Equivalent

VZEROALL:     `_mm256_zeroall()`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Table 2-25, "Type 8 Class Exception Conditions."

## VZEROUPPER—Zero Upper Bits of YMM and ZMM Registers

| Opcode/<br>Instruction          | Op<br>/<br>En | 64/32<br>bit Mode<br>Support | CPUID<br>Feature<br>Flag | Description  |
|---------------------------------|---------------|------------------------------|--------------------------|--|
| VEX.128.OF.WIG 77<br>VZEROUPPER | ZO            | V/V                          | AVX                      | Zero bits in positions 128 and higher of some YMM and ZMM registers. |

### Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|-------|-----------|-----------|-----------|-----------|
| ZO    | N/A       | N/A       | N/A       | N/A       |

### Description

In 64-bit mode, the instruction zeroes the bits in positions 128 and higher in YMM0-YMM15 and ZMM0-ZMM15. Outside 64-bit mode, it zeroes those bits only in YMM0-YMM7 and ZMM0-ZMM7. VZEROUPPER does not modify the lower 128 bits of these registers and it does not modify ZMM16-ZMM31.

This instruction is recommended when transitioning between AVX and legacy SSE code; it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

### Operation

`simd_reg_file[][]` is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm, and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0, and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits.

### VZEROUPPER

IF (64-bit mode)

  limit := 15

ELSE

  limit := 7

FOR i in 0 .. limit:

`simd_reg_file[i][MAXVL-1:128] := 0`

### Intel C/C++ Compiler Intrinsic Equivalent

VZEROUPPER: `_mm256_zeroupper()`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Table 2-25, "Type 8 Class Exception Conditions."

