

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

## Volume 2C: Instruction Set Reference, V-Z

**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-Z*, Order Number 326018; *Instruction Set Reference*, 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: 326018-069US  
January 2019

Intel technologies features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Learn more at [intel.com](http://intel.com), or from the OEM or retailer.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

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.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

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.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting <http://www.intel.com/design/literature.htm>.

Intel, the Intel logo, Intel Atom, Intel Core, Intel SpeedStep, MMX, Pentium, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

\*Other names and brands may be claimed as the property of others.

## 5.1 TERNARY BIT VECTOR LOGIC TABLE

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

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

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

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

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

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

- Constants: TRUE (1), FALSE (0);
- Unary function: Not (!);
- Binary functions: and, nand, or, nor, xor, xnor;
- Conditional function: Select (?:);
- Tertiary functions: major, minor.

Table 5-1. Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

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. Low 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-Z)

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

## VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

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

### Instruction Operand Encoding

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

### 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[j+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] &gt;&gt; (64\*SHIFT)

; Apply writemask

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[j+63:i] ← tmp[j+63:i]

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**

```

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 Exceptions Type E4NF.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 65 /r VBLENDMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL 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 VBLENDMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL 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 VBLENDMPD zmm1 {k1}{z}, 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 VBLENDMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL 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 VBLENDMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL 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 VBLENDMPS zmm1 {k1}{z}, 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	ModRM:r/m (r)	NA

### 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

DEST[i+63:i] ← 0 ; zeroing-masking

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

DEST[i+31:i] ← 0 ; zeroing-masking

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 Exceptions Type E4.

## VBROADCAST—Load with Broadcast Floating-Point Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r 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 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 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 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 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 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 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 VBROADCASTSD ymm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL 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 VBROADCASTSD zmm1 {k1}{z}, 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 VBROADCASTF32X2 ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL 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 VBROADCASTF32X2 zmm1 {k1}{z}, 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 VBROADCASTSS xmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL 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 VBROADCASTSS ymm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL 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 VBROADCASTSS zmm1 {k1}{z}, 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 VBROADCASTF32X4 ymm1 {k1}{z}, m128	D	V/V	AVX512VL 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 VBROADCASTF32X4 zmm1 {k1}{z}, 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 VBROADCASTF64X2 ymm1 {k1}{z}, m128	C	V/V	AVX512VL AVX512DQ	Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in ymm1 using writemask k1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 1A /r VBROADCASTF64X2 zmm1 {k1}{z}, 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 VBROADCASTF32X8 zmm1 {k1}{z}, 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 VBROADCASTF64X4 zmm1 {k1}{z}, 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	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

#### 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.

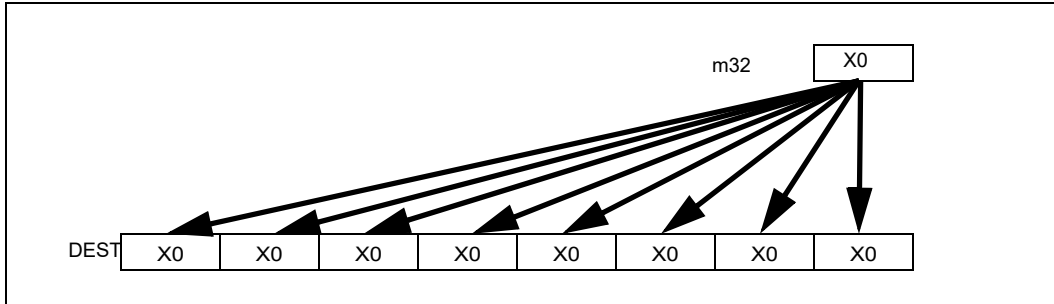


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

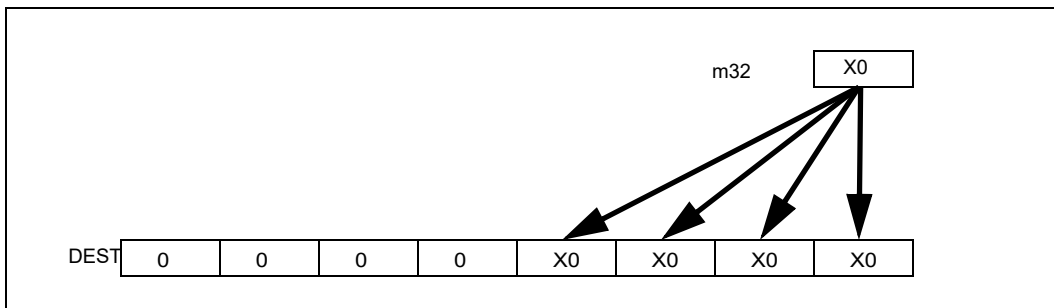


Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

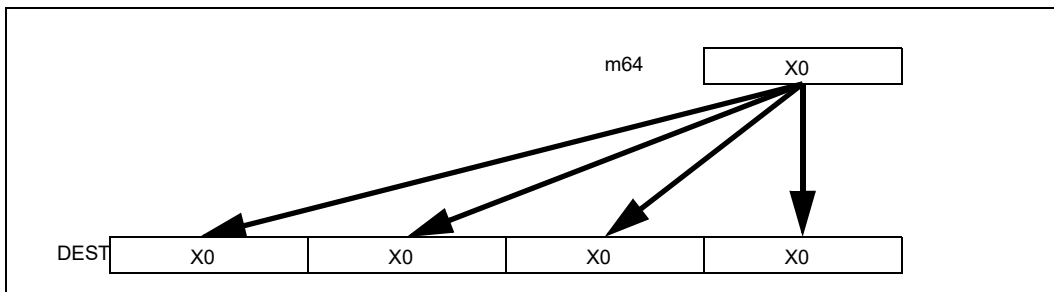


Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

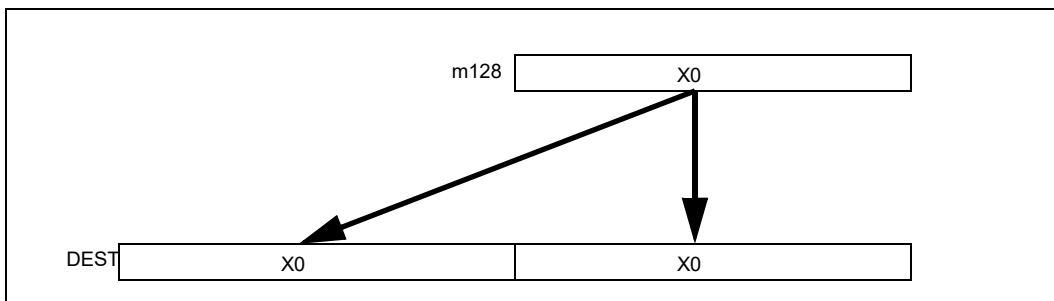


Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)



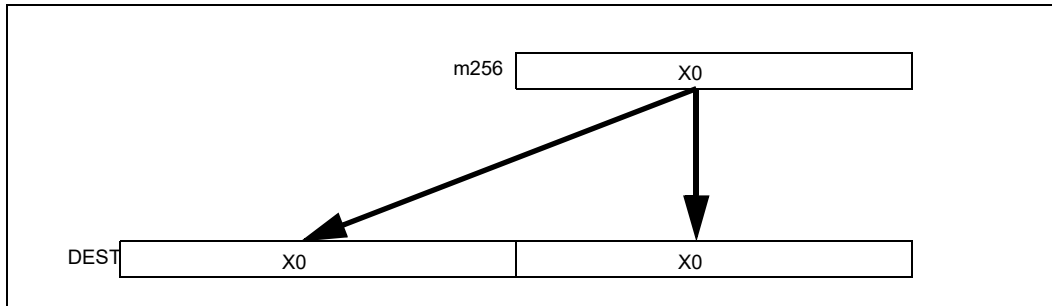


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

### 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 Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6.

#UD                    If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.  
                           If EVEX.L'L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.  
                           If EVEX.L'L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.

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

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

### 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.

#UD                   If EVEX.vvvv != 1111B.

**VCOMPRESSPS—Store Sparse Packed Single-Precision Floating-Point Values into Dense Memory**

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

**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.

#UD                    If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7B /r VCVTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VCVTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VCVTPD2QQ zmm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

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

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

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

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

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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 Exceptions Type E2

#UD                    If EVEX.vvvv != 1111B.

## VCVTPD2UDQ—Convert 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 79 /r VCVTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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.256.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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.512.0F.W1 79 /r VCVTPD2UDQ ymm1 {k1}{z}, 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)	NA	NA

### 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

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

THEN

DEST[i+31:i] ←

Convert\_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 Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 79 /r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VCVTPD2UQQ zmm1 {k1}{z}, 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)	NA	NA

### 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

Convert\_Double\_Precision\_Floating\_Point\_To\_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_cvtpd_epu64( __m512d a);
VCVTPD2UQQ __m512i __mm512_mask_cvtpd_epu64( __m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i __mm512_maskz_cvtpd_epu64( __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i __mm512_cvt_roundpd_epu64( __m512d a, int r);
VCVTPD2UQQ __m512i __mm512_mask_cvt_roundpd_epu64( __m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i __mm512_maskz_cvt_roundpd_epu64( __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i __mm256_mask_cvtpd_epu64( __m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i __mm256_maskz_cvtpd_epu64( __mmask8 k, __m256d a);
VCVTPD2UQQ __m128i __mm_mask_cvtpd_epu64( __m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i __mm_maskz_cvtpd_epu64( __mmask8 k, __m128d a);
VCVTPD2UQQ __m256i __mm256_cvtpd_epu64( __m256d src)
VCVTPD2UQQ __m128i __mm_cvtpd_epu64( __m128d src)

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2

#UD                    If EVEX.vvvv != 1111B.

## VCVTPH2PS—Convert 16-bit FP values to Single-Precision FP values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64	A	V/V	F16C	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.
VEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128	A	V/V	F16C	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.
EVEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point values in xmm1.
EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point values in ymm1.
EVEX.512.66.0F38.W0 13 /r VCVTPH2PS zmm1 {k1}{z}, ymm2/m256 {sae}	B	V/V	AVX512F	Convert sixteen packed half precision (16-bit) floating-point values in ymm2/m256 to packed single-precision floating-point values in zmm1.

### Instruction Operand Encoding

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

### Description

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

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

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

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

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

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

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

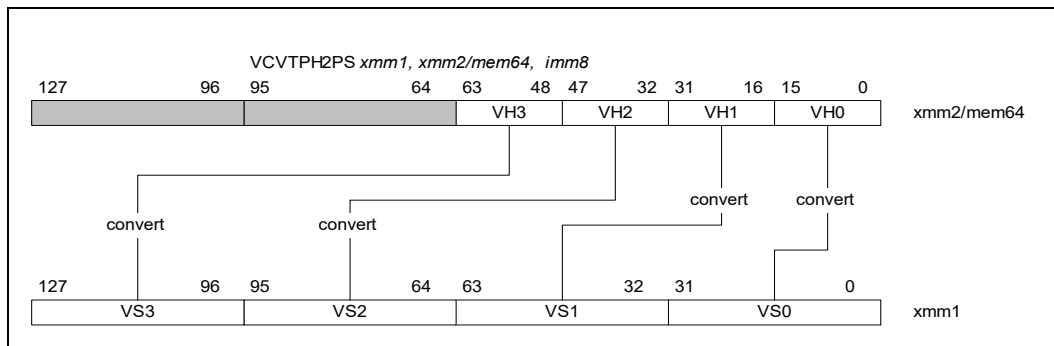


Figure 5-6. VCVTPH2PS (128-bit Version)

**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

**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)

**SIMD Floating-Point Exceptions**

Invalid

**Other Exceptions**

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

EVEX-encoded instructions, see Exceptions Type E11.

#UD If VEX.W=1.

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

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F3A.W0 1D /r ib VCVTSP2PH xmm1/m64, xmm2, 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 VCVTSP2PH xmm1/m128, ymm2, 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 VCVTSP2PH xmm1/m64 {k1}{z}, xmm2, imm8	B	V/V	AVX512VL 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 VCVTSP2PH xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	AVX512VL 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 VCVTSP2PH ymm1/m256 {k1}{z}, 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	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Half Mem	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

#### 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.

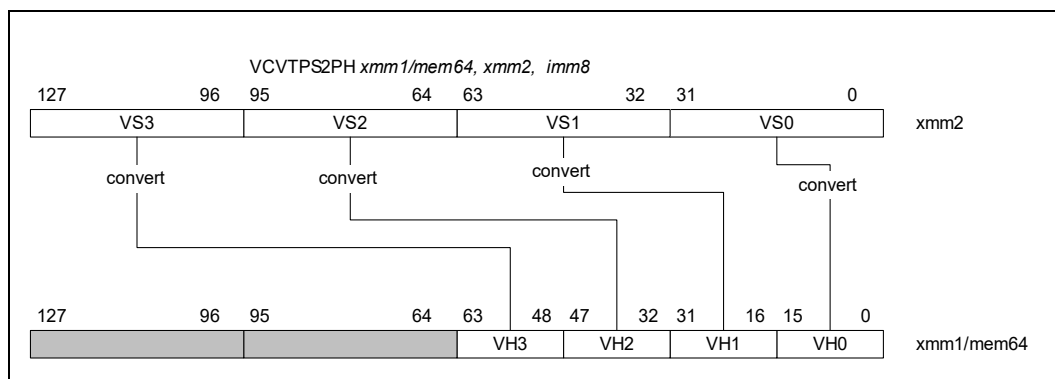


Figure 5-7. VCVTSP2PH (128-bit Version)

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 Exceptions Type 11 (do not report #AC);

EVEX-encoded instructions, see Exceptions Type E11.

#UD                    If VEX.W=1.

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

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W0 79 /r VCVTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL 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 VCVTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL 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 VCVTPS2UDQ zmm1 {k1}{z}, 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)	NA	NA

### 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****VCVTSP2UDQ (EVEX encoded versions) when src operand is a register**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 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

**VCVTSP2UDQ (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(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 Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.

## VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Singed Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7B /r VCVTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL 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 VCVTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL 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 VCVTPS2QQ zmm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

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

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

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

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

```

    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 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;
    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 Exceptions Type E3

#UD If EVEX.vvvv != 1111B.

## VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 79 /r VCVTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL 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 VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL 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 VCVTPS2UQQ zmm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

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



**Operation****VCVTSP2UQQ (EVEX encoded versions) when src operand is a register**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 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**

```

VCVTPS2UQQ __m512i __mm512_cvtps_epu64( __m512 a);
VCVTPS2UQQ __m512i __mm512_mask_cvtps_epu64( __m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i __mm512_maskz_cvtps_epu64( __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i __mm512_cvt_roundps_epu64( __m512 a, int r);
VCVTPS2UQQ __m512i __mm512_mask_cvt_roundps_epu64( __m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i __mm512_maskz_cvt_roundps_epu64( __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i __mm256_cvtps_epu64( __m256 a);
VCVTPS2UQQ __m256i __mm256_mask_cvtps_epu64( __m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i __mm256_maskz_cvtps_epu64( __mmask8 k, __m256 a);
VCVTPS2UQQ __m128i __mm_cvtps_epu64( __m128 a);
VCVTPS2UQQ __m128i __mm_mask_cvtps_epu64( __m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i __mm_maskz_cvtps_epu64( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3

#UD                    If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 E6 /r VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VCVTQQ2PD zmm1 {k1}{z}, 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)	NA	NA

### 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

Convert\_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 Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VCVTQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VCVTQQ2PS ymm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

### Operation

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

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

```

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

```

**VCVTQQ2PS (EVEX encoded versions) when src operand is a memory source**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

FOR j ← 0 TO KL-1
  i ← j * 64
  k ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[k+31:k] ←
          Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
        ELSE
          DEST[k+31:k] ←
          Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[j+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_roundepsi64_ps( __m512i a, int r);
VCVTQQ2PS __m256 __mm512_mask_cvt_roundepsi64_ps( __m256 s, __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m256 __mm512_maskz_cvt_roundepsi64_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 Exceptions Type E2

#UD If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.OF.W0 79 /r 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.LIG.F2.OF.W1 79 /r 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 instructions 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)	NA	NA

### Description

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

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

### Operation

#### VCVTSD2USI (EVEX encoded version)

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode and OperandSize = 64

THEN DEST[63:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[63:0]);

ELSE DEST[31:0] ← Convert\_Double\_Precision\_Floating\_Point\_To\_UInteger(SRC[63:0]);

FI

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

VCVTSD2USI unsigned int \_\_mm\_cvtsd\_u32(\_\_m128d);

VCVTSD2USI unsigned int \_\_mm\_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 Exceptions Type E3NF.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.OF.W0 79 /r 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.LIG.F3.OF.W1 79 /r 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 instructions 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)	NA	NA

### 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger(SRC[31:0]);

FI;

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

VCVTSS2USI unsigned \_\_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 Exceptions Type E3NF.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7A /r VCVTTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VCVTTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VCVTTPD2QQ zmm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

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

### Operation

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

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

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

$i \leftarrow j * 64$

  IF  $k1[j]$  OR \*no writemask\*

    THEN  $DEST[i+63:i] \leftarrow$

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

  ELSE

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

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

    ELSE ; zeroing-masking

$DEST[i+63:i] \leftarrow 0$

  FI

FI;

ENDFOR

$DEST[MAXVL-1:VL] \leftarrow 0$

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

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN
      IF (EVEX.b == 1)
        THEN
          DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[63:0])
        ELSE
          DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
        FI;
      ELSE
        IF *merging-masking* ; merging-masking
          THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
      FI;
    ENDFOR
  DEST[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 Exceptions Type E2.

#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 VCVTTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VCVTTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VCVTTPD2UDQ ymm1 {k1}{z}, 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)	NA	NA

### Description

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

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

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (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 Exceptions Type E2.

#UD                      If EVEX.vvvv != 1111B.

## VCVTTPD2UQQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 78 /r VCVTTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VCVTTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VCVTTPD2UQQ zmm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

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

### Operation

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

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

FOR j ← 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[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**

```

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 Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.



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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.OF.W0 78 /r VCVTTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL 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 VCVTTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL 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 VCVTTPS2UDQ zmm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

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

### Operation

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

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+31:i] ←

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

  ELSE

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

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

    ELSE ; zeroing-masking

      DEST[i+31:i] ← 0

  FI

FI;

ENDFOR

DEST[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_cvt_roundps_epu32( __m512 a, int sae);
VCVTTPS2UDQ __m512i __mm512_mask_cvt_roundps_epu32( __m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i __mm512_maskz_cvt_roundps_epu32( __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i __mm256_mask_cvttps_epu32( __m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i __mm256_maskz_cvttps_epu32( __mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i __mm_mask_cvttps_epu32( __m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i __mm_maskz_cvttps_epu32( __mmask8 k, __m128 a);

```

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Singed Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7A /r VCVTTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL 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 VCVTTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL 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 VCVTTPS2QQ zmm1 {k1}{z}, 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)	NA	NA

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

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

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

### Operation

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

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

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

$i \leftarrow j * 64$

$k \leftarrow j * 32$

    IF  $k1[j]$  OR \*no writemask\*

        THEN  $DEST[i+63:i] \leftarrow$

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

    ELSE

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

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

        ELSE ; zeroing-masking

$DEST[i+63:i] \leftarrow 0$

    FI

FI;

ENDFOR

$DEST[MAXVL-1:VL] \leftarrow 0$

**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 Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 78 /r VCVTTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL 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 VCVTTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL 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 VCVTTPS2UQQ zmm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

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

### Operation

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

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

FOR j ← 0 TO KL-1

    i ← j \* 64

    k ← j \* 32

    IF k1[j] OR \*no writemask\*

        THEN DEST[i+63: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]]_cvtt[_round]ps_epu64
VCVTTPS2UQQ __m512i __mm512_cvttps_epu64( __m256 a);
VCVTTPS2UQQ __m512i __mm512_mask_cvttps_epu64( __m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_maskz_cvttps_epu64( __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i __mm512_cvtt_roundps_epu64( __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_mask_cvtt_roundps_epu64( __m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m512i __mm512_maskz_cvtt_roundps_epu64( __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m256i __mm256_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 Exceptions Type E3.

#UD If EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.OF.W0 78 /r 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.LIG.F2.OF.W1 78 /r 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:

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

### Instruction Operand Encoding

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

### Description

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

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

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

### Operation

#### VCVTTSD2USI (EVEX encoded version)

IF 64-Bit Mode and OperandSize = 64

```
THEN  DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
ELSE  DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
```

FI

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

```
VCVTTSD2USI unsigned int __mm_cvttstd_u32(__m128d);
VCVTTSD2USI unsigned int __mm_cvtt_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 __mm_cvttstd_u64(__m128d);
VCVTTSD2USI unsigned __int64 __mm_cvtt_roundsd_u64(__m128d, int sae);
```

#### SIMD Floating-Point Exceptions

Invalid, Precision

#### Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

## VCVTTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.OF.W0 78 /r 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.LIG.F3.OF.W1 78 /r 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:

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

### Instruction Operand Encoding

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

### Description

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

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

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

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



**Operation****VCVTTSS2USI (EVEX encoded version)**

IF 64-bit Mode and OperandSize = 64

THEN

DEST[63:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[31:0]);

ELSE

DEST[31:0] ← Convert\_Single\_Precision\_Floating\_Point\_To\_UInteger\_Truncate(SRC[31:0]);

FI;

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

VCVTTSS2USI unsigned int \_mm\_cvttss\_u32( \_\_m128 a);

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

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

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

**SIMD Floating-Point Exceptions**

Invalid, Precision

**Other Exceptions**

EVEX-encoded instructions, see Exceptions Type E3NF.

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

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

### Instruction Operand Encoding

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

### Description

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

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

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

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

### Operation

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

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

FOR j ← 0 TO KL-1

  i ← j \* 64

  k ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN DEST[i+63:i] ←

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

  ELSE

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

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

    ELSE ; zeroing-masking

      DEST[i+63:i] ← 0

  FI

FI;

ENDFOR

DEST[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 Exceptions Type E5.

#UD If EVEX.vvvv != 1111B.

## VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W0 7A /r VCVTUDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL 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 VCVTUDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL 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 VCVTUDQ2PS zmm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

### Operation

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

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN DEST[i+31:i] ←

Convert\_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 Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 7A /r VCVTUQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VCVTUQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VCVTUQQ2PD zmm1 {k1}{z}, 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)	NA	NA

#### Description

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

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

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

#### Operation

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

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

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

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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 Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single-precision floating-point values in zmm1 with writemask k1.
EVEX.256.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.F2.0F.W1 7A /r VCVTUQQ2PS ymm1 {k1}{z}, 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)	NA	NA

### Description

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

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

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

### Operation

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

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

k ← j \* 64

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

THEN DEST[i+31:i] ←

Convert\_UQuadInteger\_To\_Single\_Precision\_Floating\_Point(SRC[k+63:k])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[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 Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

**VCVTUSI2SD—Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.OF.W0 7B /r 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.LIG.F2.OF.W1 7B /r 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:**

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

**Instruction Operand Encoding**

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

**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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[63:0] ← Convert\_UInteger\_To\_Double\_Precision\_Floating\_Point(SRC2[63:0]);

ELSE

DEST[63:0] ← Convert\_UInteger\_To\_Double\_Precision\_Floating\_Point(SRC2[31:0]);

FI;

DEST[127:64] ← SRC1[127:64]

DEST[MAXVL-1:128] ← 0

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

VCVTUSI2SD \_\_m128d \_mm\_cvtsd( \_\_m128d s, unsigned a);

VCVTUSI2SD \_\_m128d \_mm\_cvtsd( \_\_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 Exceptions Type E3NF if W1, else type E10NF.

## VCVTUSI2SS—Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.OF.W0 7B /r 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.LIG.F3.OF.W1 7B /r 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:

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

### Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF 64-Bit Mode And OperandSize = 64

THEN

DEST[31:0] ← Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[63:0]);

ELSE

DEST[31:0] ← Convert\_UInteger\_To\_Single\_Precision\_Floating\_Point(SRC[31:0]);

FI;

DEST[127:32] ← SRC1[127:32]

DEST[MAXVL-1:128] ← 0

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

VCVTUSI2SS \_\_m128 \_\_mm\_cvts32\_ss( \_\_m128 s, unsigned a);

VCVTUSI2SS \_\_m128 \_\_mm\_cvt\_roundu32\_ss( \_\_m128 s, unsigned a, int r);

VCVTUSI2SS \_\_m128 \_\_mm\_cvts64\_ss( \_\_m128 s, unsigned \_\_int64 a);

VCVTUSI2SS \_\_m128 \_\_mm\_cvt\_roundu64\_ss( \_\_m128 s, unsigned \_\_int64 a, int r);

## **SIMD Floating-Point Exceptions**

Precision

## **Other Exceptions**

See Exceptions Type E3NF.

## VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 42 /r ib VDBPSADBW xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL 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 VDBPSADBW ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL 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 VDBPSADBW zmm1 {k1}{z}, zmm2, 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	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.

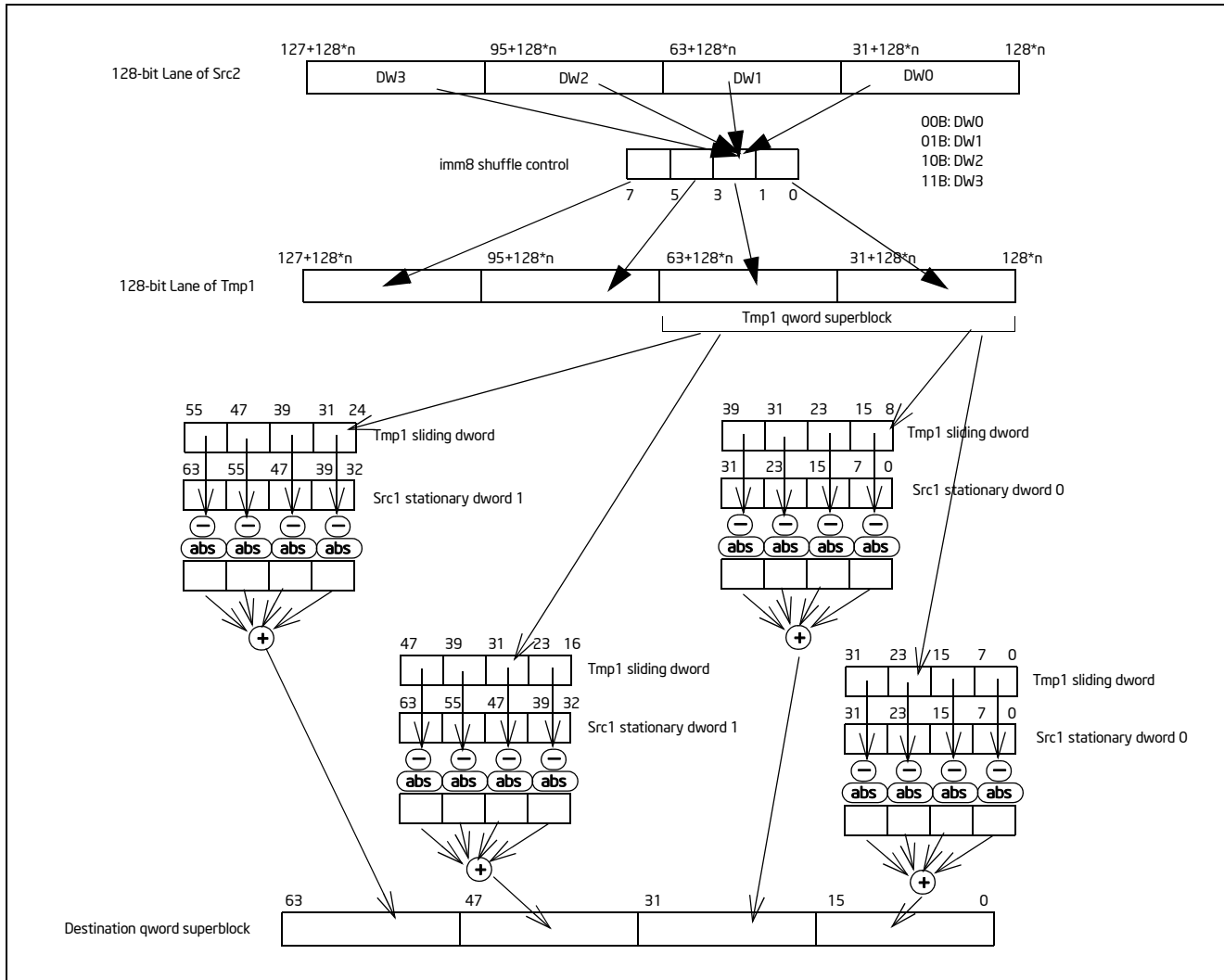


Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBW

**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

$$\begin{aligned} \text{TMP\_DEST}[I+15:I] &\leftarrow \text{ABS}(\text{SRC1}[I+7: I] - \text{TMP1}[I+7: I]) + \\ &\quad \text{ABS}(\text{SRC1}[I+15: I+8] - \text{TMP1}[I+15: I+8]) + \\ &\quad \text{ABS}(\text{SRC1}[I+23: I+16] - \text{TMP1}[I+23: I+16]) + \\ &\quad \text{ABS}(\text{SRC1}[I+31: I+24] - \text{TMP1}[I+31: I+24]) \end{aligned}$$

$$\begin{aligned} \text{TMP\_DEST}[I+31: I+16] &\leftarrow \text{ABS}(\text{SRC1}[I+7: I] - \text{TMP1}[I+15: I+8]) + \\ &\quad \text{ABS}(\text{SRC1}[I+15: I+8] - \text{TMP1}[I+23: I+16]) + \\ &\quad \text{ABS}(\text{SRC1}[I+23: I+16] - \text{TMP1}[I+31: I+24]) + \\ &\quad \text{ABS}(\text{SRC1}[I+31: I+24] - \text{TMP1}[I+39: I+32]) \end{aligned}$$

$$\begin{aligned} \text{TMP\_DEST}[I+47: I+32] &\leftarrow \text{ABS}(\text{SRC1}[I+39: I+32] - \text{TMP1}[I+23: I+16]) + \\ &\quad \text{ABS}(\text{SRC1}[I+47: I+40] - \text{TMP1}[I+31: I+24]) + \\ &\quad \text{ABS}(\text{SRC1}[I+55: I+48] - \text{TMP1}[I+39: I+32]) + \\ &\quad \text{ABS}(\text{SRC1}[I+63: I+56] - \text{TMP1}[I+47: I+40]) \end{aligned}$$

$$\begin{aligned} \text{TMP\_DEST}[I+63: I+48] &\leftarrow \text{ABS}(\text{SRC1}[I+39: I+32] - \text{TMP1}[I+31: I+24]) + \\ &\quad \text{ABS}(\text{SRC1}[I+47: I+40] - \text{TMP1}[I+39: I+32]) + \\ &\quad \text{ABS}(\text{SRC1}[I+55: I+48] - \text{TMP1}[I+47: I+40]) + \\ &\quad \text{ABS}(\text{SRC1}[I+63: I+56] - \text{TMP1}[I+55: I+48]) \end{aligned}$$

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);  
 VDBPSADBW \_\_m512i\_mm512\_mask\_dbsad\_epu8(\_\_m512i s, \_\_mmask32 m, \_\_m512i a, \_\_m512i b);  
 VDBPSADBW \_\_m512i\_mm512\_maskz\_dbsad\_epu8(\_\_mmask32 m, \_\_m512i a, \_\_m512i b);  
 VDBPSADBW \_\_m256i\_mm256\_dbsad\_epu8(\_\_m256i a, \_\_m256i b);  
 VDBPSADBW \_\_m256i\_mm256\_mask\_dbsad\_epu8(\_\_m256i s, \_\_mmask16 m, \_\_m256i a, \_\_m256i b);  
 VDBPSADBW \_\_m256i\_mm256\_maskz\_dbsad\_epu8(\_\_mmask16 m, \_\_m256i a, \_\_m256i b);  
 VDBPSADBW \_\_m128i\_mm\_dbsad\_epu8(\_\_m128i a, \_\_m128i b);  
 VDBPSADBW \_\_m128i\_mm\_mask\_dbsad\_epu8(\_\_m128i s, \_\_mmask8 m, \_\_m128i a, \_\_m128i b);  
 VDBPSADBW \_\_m128i\_mm\_maskz\_dbsad\_epu8(\_\_mmask8 m, \_\_m128i a, \_\_m128i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4NF.nb.

**VEXPANDPD—Load Sparse Packed Double-Precision Floating-Point Values from Dense Memory**

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 88 /r VEXPANDPD xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 88 /r VEXPANDPD ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 88 /r 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)	NA	NA

**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)

$k \leftarrow 0$

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

$i \leftarrow j * 64$

    IF  $k1[j]$  OR \*no writemask\*

        THEN

$DEST[i+63:i] \leftarrow SRC[k+63:k];$

$k \leftarrow k + 64$

        ELSE

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

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

                ELSE ; zeroing-masking

                    THEN  $DEST[i+63:i] \leftarrow 0$

        FI

    FI;

ENDFOR

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

**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.

#UD                    If EVEX.vvvv != 1111B.

## VEXPANDPS—Load Sparse Packed Single-Precision Floating-Point Values from Dense Memory

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 88 /r VEXPANDPS xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 88 /r 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)	NA	NA

#### 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.

#UD                    If EVEX.vvvv != 1111B.

**VERR/VERW—Verify a Segment for Reading or Writing**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 00 /4	VERR <i>r/m16</i>	M	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be read.
OF 00 /5	VERW <i>r/m16</i>	M	Valid	Valid	Set ZF=1 if segment specified with <i>r/m16</i> can be written.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA	NA

**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;
```

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

## Real-Address Mode Exceptions

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

## Virtual-8086 Mode Exceptions

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

## Compatibility Mode Exceptions

Same exceptions as in protected mode.

## 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

## VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEXTRACTF128 xmm1/m128, ymm2, 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 VEXTRACTF32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	AVX512VL 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 VEXTRACTF32x4 xmm1/m128 {k1}{z}, 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 VEXTRACTF64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	AVX512VL 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 VEXTRACTF64X2 xmm1/m128 {k1}{z}, 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 VEXTRACTF32X8 ymm1/m256 {k1}{z}, 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 VEXTRACTF64x4 ymm1/m256 {k1}{z}, 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	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

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

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

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

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

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

The high 6 bits of the immediate are ignored.

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



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

VL = 256, 512

IF VL = 256

```

CASE (imm8[0]) OF
  0: TMP_DEST[127:0] ← SRC1[127:0]
  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

**VEEXTRACTF64x2 (EVEX encoded versions) when destination is a register**

VL = 256, 512

IF VL = 256

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

FI;

IF VL = 512

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

FI;

FOR j ← 0 TO 1

i ← j \* 64

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

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

ELSE

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

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

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:128] ← 0

**VEEXTRACTF64x2 (EVEX encoded versions) when destination is memory**

VL = 256, 512

IF VL = 256

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

FI;

IF VL = 512

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

FI;

FOR j ← 0 TO 1

i ← j \* 64

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

THEN DEST[i+63:i] ← TMP\_DEST[i+63:i]

```

        ELSE *DEST[i+63:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR

```

#### **VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is a register**

VL = 512

```

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

```

```

FOR j ← 0 TO 7
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
    ELSE
        IF *merging-masking*      ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
        ELSE *zeroing-masking*    ; zeroing-masking
            DEST[i+31:i] ← 0
        FI
    FI;
ENDFOR
DEST[MAXVL-1:256] ← 0

```

#### **VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is memory**

```

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

```

```

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

```

#### **VEXTRACTF64x4 (EVEX.512 encoded version) when destination is a register**

VL = 512

```

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

```

```

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

```

```

                DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] ← 0

```

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

```

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

```

```

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

```

**VEXTRACTF128 (memory destination form)**

```

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

```

**VEXTRACTF128 (register destination form)**

```

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

```

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

```

VEXTRACTF32x4 __m128 _mm512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_extractf32x4_ps(__m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_maskz_extractf32x4_ps(__mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_extractf32x8_ps(__m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_mask_extractf32x8_ps(__m256 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_maskz_extractf32x8_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_extractf64x2_pd(__m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_maskz_extractf64x2_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_extractf64x2_pd(__m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_maskz_extractf64x2_pd(__mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_extractf64x4_pd(__m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_mask_extractf64x4_pd(__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 Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.

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

## VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 39 /r ib VEXTRACTI128 xmm1/m128, ymm2, 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 VEXTRACTI32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	AVX512VL 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 VEXTRACTI32x4 xmm1/m128 {k1}{z}, 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 VEXTRACTI64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	AVX512VL 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 VEXTRACTI64X2 xmm1/m128 {k1}{z}, 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 VEXTRACTI32X8 ymm1/m256 {k1}{z}, 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 VEXTRACTI64x4 ymm1/m256 {k1}{z}, 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	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

### Description

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

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

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

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

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

**VEEXTRACTI64x4:** The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

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

The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.

If 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 (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

#### **VEEXTRACTI32x4 (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

```

**VEEXTRACTI64x2 (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



**VEEXTRACTI64x2 (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

**VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory**

```

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

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

```

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

```

VL = 512
CASE (imm8[0]) OF
  0: TMP_DEST[255:0] ← SRC1[255:0]
  1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
    ELSE
      IF *merging-masking*      ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE *zeroing-masking*    ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
ENDFOR
DEST[MAXVL-1:256] ← 0

```

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

```

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

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

```

**VEXTRACTI128 (memory destination form)**

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

**VEXTRACTI128 (register destination form)**

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

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

VEXTRACTI32x4 \_\_m128i\_mm512\_extracti32x4\_epi32(\_\_m512i a, const int nidx);  
 VEXTRACTI32x4 \_\_m128i\_mm512\_mask\_extracti32x4\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI32x4 \_\_m128i\_mm512\_maskz\_extracti32x4\_epi32(\_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI32x4 \_\_m128i\_mm256\_extracti32x4\_epi32(\_\_m256i a, const int nidx);  
 VEXTRACTI32x4 \_\_m128i\_mm256\_mask\_extracti32x4\_epi32(\_\_m128i s, \_\_mmask8 k, \_\_m256i a, const int nidx);  
 VEXTRACTI32x4 \_\_m128i\_mm256\_maskz\_extracti32x4\_epi32(\_\_mmask8 k, \_\_m256i a, const int nidx);  
 VEXTRACTI32x8 \_\_m256i\_mm512\_extracti32x8\_epi32(\_\_m512i a, const int nidx);  
 VEXTRACTI32x8 \_\_m256i\_mm512\_mask\_extracti32x8\_epi32(\_\_m256i s, \_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI32x8 \_\_m256i\_mm512\_maskz\_extracti32x8\_epi32(\_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI64x2 \_\_m128i\_mm512\_extracti64x2\_epi64(\_\_m512i a, const int nidx);  
 VEXTRACTI64x2 \_\_m128i\_mm512\_mask\_extracti64x2\_epi64(\_\_m128i s, \_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI64x2 \_\_m128i\_mm512\_maskz\_extracti64x2\_epi64(\_\_mmask8 k, \_\_m512i a, const int nidx);  
 VEXTRACTI64x2 \_\_m128i\_mm256\_extracti64x2\_epi64(\_\_m256i a, const int nidx);  
 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 Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6NF.

#UD IF VEX.L = 0.

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

## VFIXUPIMMPD—Fix Up Special Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 54 /r ib VFIXUPIMMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL 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 VFIXUPIMMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL 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 VFIXUPIMMPD zmm1 {k1}{z}, zmm2, 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	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 DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

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

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

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

MXCSR 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[i+63:i], SRC2[63:0], imm8 [7:0])

ELSE

DEST[i+63:i] ← FIXUPIMM\_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[i+63:i], imm8 [7:0])

FI;

ELSE

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

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

ELSE DEST[i+63:i] ← 0 ; zeroing-masking

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

## Immediate Control Description:

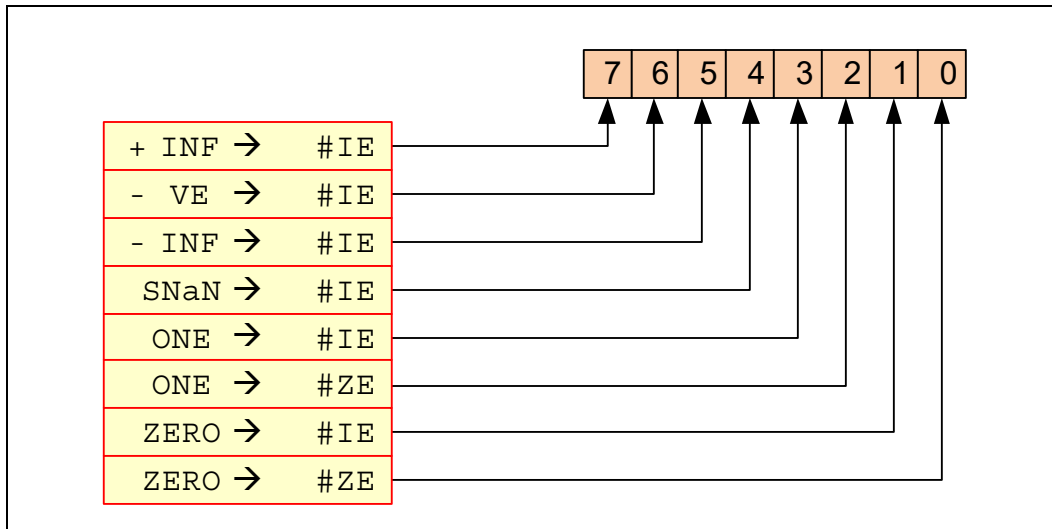


Figure 5-9. VFIXUPIMMPD Immediate Control Description

## 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, __m256i tbl, int imm);
VFIXUPIMMPD __m256d_mm256_mask_fixupimm_pd(__m256d s, __mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m256d_mm256_maskz_fixupimm_pd( __mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m128d_mm_fixupimm_pd( __m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d_mm_mask_fixupimm_pd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d_mm_maskz_fixupimm_pd( __mmask8 k, __m128d a, __m128i tbl, int imm);

```

## SIMD Floating-Point Exceptions

Zero, Invalid

## Other Exceptions

See Exceptions Type E2.

## VFIXUPIMMPS—Fix Up Special Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 54 /r VFIXUPIMMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL 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 VFIXUPIMMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL 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 VFIXUPIMMPS zmm1 {k1}{z}, zmm2, 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	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 SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , 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], tbl3[31:0], imm8 [7:0]){
  tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ = 1)) ? 0.0 : src1[31:0]
  CASE(tsrc[31:0] of TOKEN_TYPE) {
    QNAN_TOKEN: j ← 0;
    SNAN_TOKEN: j ← 1;
    ZERO_VALUE_TOKEN: j ← 2;
    POS_ONE_VALUE_TOKEN: j ← 3;
    NEG_INF_TOKEN: j ← 4;
    POS_INF_TOKEN: j ← 5;
    NEG_VALUE_TOKEN: j ← 6;
    POS_VALUE_TOKEN: j ← 7;
  } ; end source special CASE(tsrc...)
```

; The required response from src3 table is extracted

```
token_response[3:0] = tbl3[3+4*j:4*j];
```

```
CASE(token_response[3:0]) {
  0000: dest[31:0] ← dest[31:0]; ; preserve content of DEST
  0001: dest[31:0] ← tsrc[31:0]; ; pass through src1 normal input value, denormal as zero
  0010: dest[31:0] ← QNaN(tsrc[31:0]);
  0011: dest[31:0] ← QNAN_Indefinite;
  0100: dest[31:0] ← -INF;
  0101: dest[31:0] ← +INF;
  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:

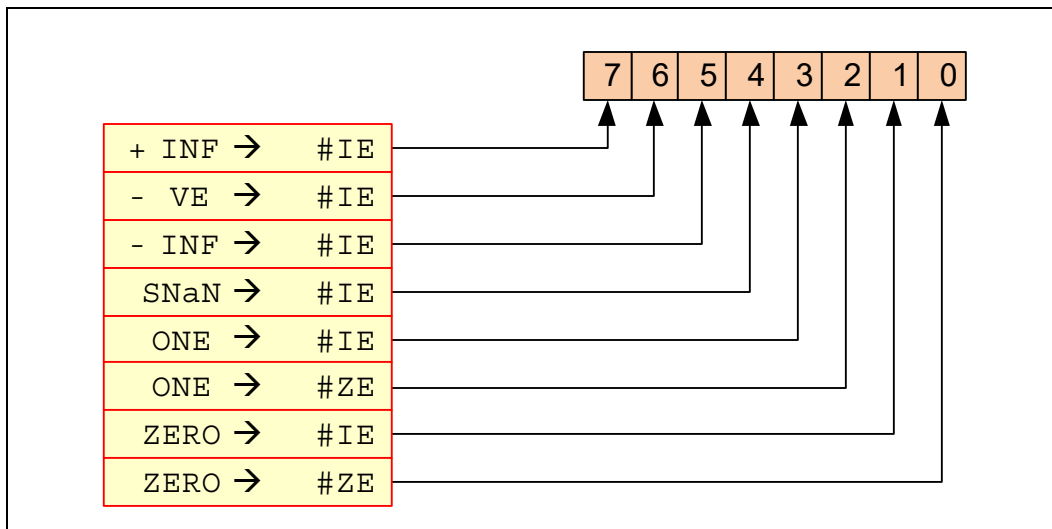


Figure 5-10. VFIXUPIMMPS Immediate Control Description

## 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, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_mask_fixupimm_ps(__m256 s, __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 __mm256_maskz_fixupimm_ps( __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m128 __mm_fixupimm_ps( __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm_mask_fixupimm_ps(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 __mm_maskz_fixupimm_ps( __mmask8 k, __m128 a, __m128i tbl, int imm);

```

## SIMD Floating-Point Exceptions

Zero, Invalid

## Other Exceptions

See Exceptions Type E2.

## VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.OF3A.W1 55 /r ib VFIXUPIMMSD xmm1 {k1}{z}, 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	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 DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x \approx 1/0$ , yields an incorrect result. To deal with this, 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:

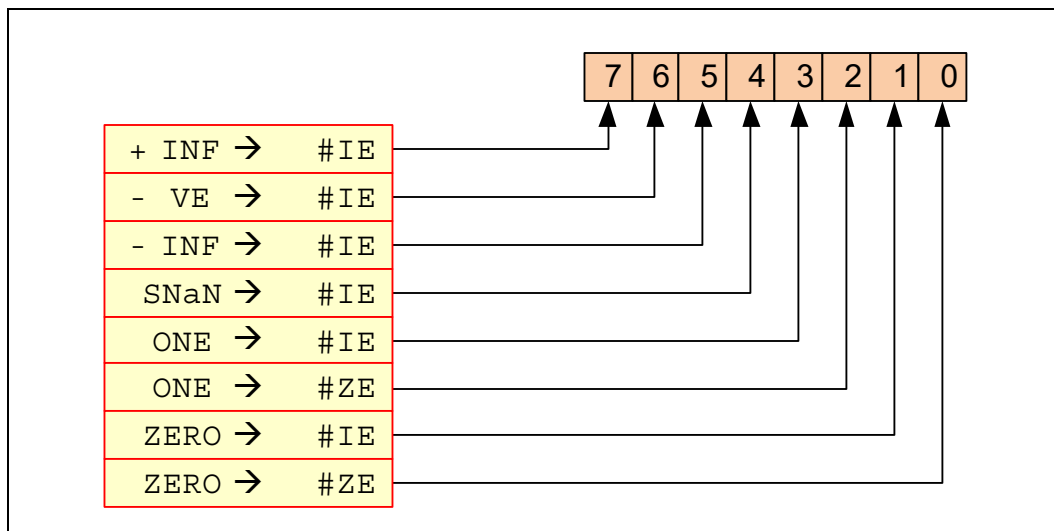


Figure 5-11. VFIXUPIMMSD Immediate Control Description

**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 Exceptions Type E3.

## VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W0 55 /r ib VFIXUPIMMSS xmm1 {k1}{z}, xmm2, 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	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 SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider `rcp(0)`. Input 0 to `rcp`, and you should get INF according to the DX10 spec. However, evaluating `rcp` via Newton-Raphson, where  $x = \text{approx}(1/0)$ , 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] ← ½;
  1100: dest[31:0] ← 90.0;
  1101: dest[31:0] ← PI/2;
  1110: dest[31:0] ← MAX_FLOAT;
  1111: dest[31:0] ← -MAX_FLOAT;
} ; end of token_response CASE

```

; The required fault reporting from imm8 is extracted

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

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

```
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
```

```
IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
```

```
 ; end fault reporting
```

```
return dest[31:0];
```

```
} ; end of FIXUPIMM_SP()
```



**VFIXUPIMMSS (EVEX encoded version)**

```

IF 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:

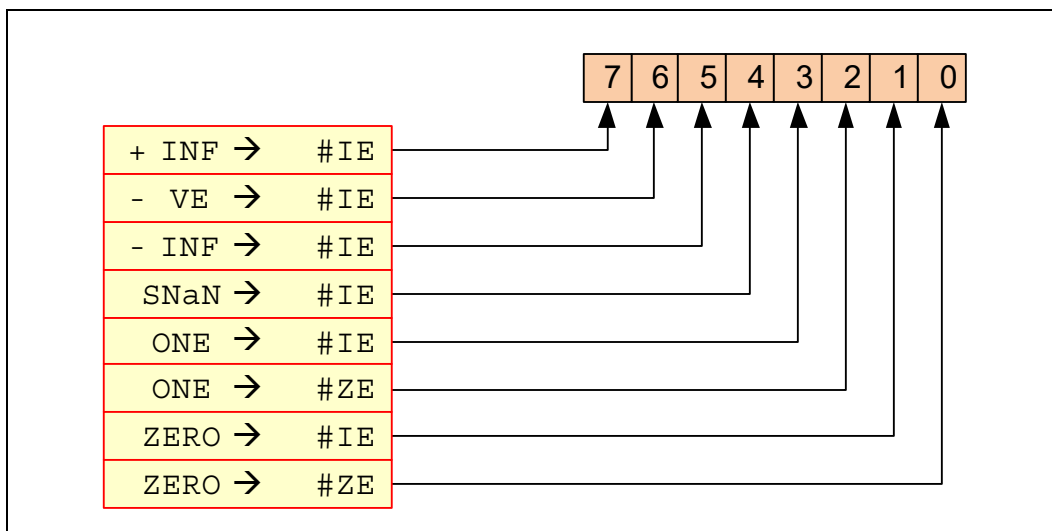


Figure 5-12. VFIXUPIMMSS Immediate Control Description

**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 Exceptions Type E3.

## VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, 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 VFMADD213PD xmm1, xmm2, 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 VFMADD231PD xmm1, xmm2, 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 VFMADD132PD ymm1, ymm2, 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 VFMADD213PD ymm1, ymm2, 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 VFMADD231PD ymm1, ymm2, 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 VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL 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 VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMADD132PD zmm1 {k1}{z}, zmm2, 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 VFMADD213PD zmm1 {k1}{z}, zmm2, 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 VFMADD231PD zmm1 {k1}{z}, zmm2, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] + SRC2[i+63:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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

**VFMAADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] + DEST[i+63:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADD231PD 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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.



## VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 98 /r VFMADD132PS xmm1, xmm2, 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 VFMADD213PS xmm1, xmm2, 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 VFMADD231PS xmm1, xmm2, 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 VFMADD132PS ymm1, ymm2, 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 VFMADD213PS ymm1, ymm2, 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.W0 B8 /r VFMADD231PS ymm1, ymm2, 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 VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMADD132PS zmm1 {k1}{z}, zmm2, 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 VFMADD213PS zmm1 {k1}{z}, zmm2, 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 VFMADD231PS zmm1 {k1}{z}, zmm2, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## 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

```

**VFMADD213PS DEST, SRC2, SRC3**

```

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

```

**VFMADD231PS DEST, SRC2, SRC3**

```

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

```

**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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN DEST[i+31:i] ←

RoundFPControl(DEST[i+31:i]\*SRC3[i+31:i] + SRC2[i+31:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADD132PS 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[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

**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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*SRC3[i+31:i] + DEST[i+31:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[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[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[31:0] + DEST[i+31:i])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[i+31:i] + DEST[i+31:i])

FI;

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

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

```

VFMADDxxxPS __m512 __mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 __mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 __mm512_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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1, xmm2, 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 VFMADD213SD xmm1, xmm2, 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 VFMADD231SD xmm1, xmm2, 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.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1 {k1}{z}, xmm2, 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.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 {k1}{z}, xmm2, 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.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 {k1}{z}, xmm2, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### 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).

**VFMAADD132SD DEST, SRC2, SRC3 (EVEX encoded version)**

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(DEST[63:0]\*SRC3[63:0] + SRC2[63:0])

ELSE

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

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAXVL-1:128] ← 0

**VFMAADD213SD DEST, SRC2, SRC3 (EVEX encoded version)**

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]\*DEST[63:0] + SRC3[63:0])

ELSE

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

THEN \*DEST[63:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[63:0] ← 0

FI;

FI;

DEST[127:64] ← DEST[127:64]

DEST[MAXVL-1:128] ← 0

**VFMAADD231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1, xmm2, 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 VFMADD213SS xmm1, xmm2, 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 VFMADD231SS xmm1, xmm2, 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.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1 {k1}{z}, xmm2, 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.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1 {k1}{z}, xmm2, 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.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1 {k1}{z}, xmm2, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### 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).

#### **VFMAADD132SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0

```

#### **VFMAADD213SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
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_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1, xmm2, 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 VFMADDSUB213PD xmm1, xmm2, 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 VFMADDSUB231PD xmm1, xmm2, 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 VFMADDSUB132PD ymm1, ymm2, 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 VFMADDSUB213PD ymm1, ymm2, 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 VFMADDSUB231PD ymm1, ymm2, 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 VFMADDSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMADDSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMADDSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMADDSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMADDSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMADDSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 A6 /r VFMADDSUB213PD zmm1 {k1}{z}, 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 VFMADDSUB231PD zmm1 {k1}{z}, 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 VFMADDSUB132PD zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

#### 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

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])$   
 $DEST[MAXVL-1:128] \leftarrow 0$

ELSEIF (VEX.256)

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])$   
 $DEST[191:128] \leftarrow RoundFPControl\_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])$   
 $DEST[255:192] \leftarrow RoundFPControl\_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192])$

FI

#### VFMADDSUB213PD DEST, SRC2, SRC3

IF (VEX.128) THEN

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])$   
 $DEST[MAXVL-1:128] \leftarrow 0$

ELSEIF (VEX.256)

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])$   
 $DEST[191:128] \leftarrow RoundFPControl\_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])$   
 $DEST[255:192] \leftarrow RoundFPControl\_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192])$

FI

#### VFMADDSUB231PD DEST, SRC2, SRC3

IF (VEX.128) THEN

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])$   
 $DEST[MAXVL-1:128] \leftarrow 0$

ELSEIF (VEX.256)

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])$   
 $DEST[191:128] \leftarrow RoundFPControl\_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])$   
 $DEST[255:192] \leftarrow RoundFPControl\_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192])$

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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] - SRC2[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] + SRC2[i+63:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMAADDSUB132PD 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;
      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 register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] - DEST[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] + DEST[i+63:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

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

```

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

```

VFMADDSUBxxxPD __m512d _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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1, xmm2, 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 VFMADDSUB213PS xmm1, xmm2, 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 VFMADDSUB231PS xmm1, xmm2, 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 VFMADDSUB132PS ymm1, ymm2, 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 VFMADDSUB213PS ymm1, ymm2, 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 VFMADDSUB231PS ymm1, ymm2, 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 VFMADDSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMADDSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMADDSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMADDSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMADDSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMADDSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMADDSUB213PS zmm1 {k1}{z}, 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 VFMADDSUB231PS zmm1 {k1}{z}, 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 VFMADDSUB132PS zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

**VFMAADDSUB132PS:** 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).

**VFMAADDSUB213PS:** 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).

**VFMAADDSUB231PS:** 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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[j+31:i] - SRC2[j+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[j+31:i] + SRC2[j+31:i])
            FI
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (4, 128), (8, 256), (16, 512)

```

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

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[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[j+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[j+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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
    FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[j+31:i] ←
                    RoundFPControl(SRC2[j+31:i]*SRC3[j+31:i] - DEST[j+31:i])
            ELSE DEST[j+31:i] ←
                RoundFPControl(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

```

**VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN
      IF j *is even*
        THEN
          IF (EVEX.b = 1)
            THEN
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
            ELSE
              DEST[i+31:i] ←
                RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[j+31:i] - DEST[i+31:i])
            FI;
          ELSE
            IF (EVEX.b = 1)
              THEN
                DEST[i+31:i] ←
                  RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
              ELSE
                DEST[i+31:i] ←
                  RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[j+31:i] + DEST[i+31:i])
              FI;
          FI
        ELSE
          IF *merging-masking* ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
          ELSE ; zeroing-masking
            DEST[i+31:i] ← 0
          FI
        FI;
      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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.



## VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1, xmm2, 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 VFMSUBADD213PD xmm1, xmm2, 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 VFMSUBADD231PD xmm1, xmm2, 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 VFMSUBADD132PD ymm1, ymm2, 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 VFMSUBADD213PD ymm1, ymm2, 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 VFMSUBADD231PD ymm1, ymm2, 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 VFMSUBADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMSUBADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMSUBADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMSUBADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMSUBADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMSUBADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 97 /r VFMSUBADD132PD zmm1 {k1}{z}, 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 VFMSUBADD213PD zmm1 {k1}{z}, 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 VFMSUBADD231PD zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

#### 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

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])$   
 $DEST[MAXVL-1:128] \leftarrow 0$

ELSEIF (VEX.256)

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])$   
 $DEST[191:128] \leftarrow RoundFPControl\_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])$   
 $DEST[255:192] \leftarrow RoundFPControl\_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192])$

FI

#### VFMSUBADD213PD DEST, SRC2, SRC3

IF (VEX.128) THEN

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])$   
 $DEST[MAXVL-1:128] \leftarrow 0$

ELSEIF (VEX.256)

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])$   
 $DEST[191:128] \leftarrow RoundFPControl\_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])$   
 $DEST[255:192] \leftarrow RoundFPControl\_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192])$

FI

#### VFMSUBADD231PD DEST, SRC2, SRC3

IF (VEX.128) THEN

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])$   
 $DEST[MAXVL-1:128] \leftarrow 0$

ELSEIF (VEX.256)

$DEST[63:0] \leftarrow RoundFPControl\_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])$   
 $DEST[127:64] \leftarrow RoundFPControl\_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])$   
 $DEST[191:128] \leftarrow RoundFPControl\_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])$   
 $DEST[255:192] \leftarrow RoundFPControl\_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192])$

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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] + SRC2[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] - SRC2[i+63:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] + SRC3[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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[j+63:i] ←
              RoundFPControl_MXCSR(SRC2[j+63:i]*DEST[j+63:i] + SRC3[63:0])
            ELSE
              DEST[j+63:i] ←
              RoundFPControl_MXCSR(SRC2[j+63:i]*DEST[j+63:i] + SRC3[j+63:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[j+63:i] ←
              RoundFPControl_MXCSR(SRC2[j+63:i]*DEST[j+63:i] - SRC3[63:0])
            ELSE
              DEST[j+63:i] ←
              RoundFPControl_MXCSR(SRC2[j+63:i]*DEST[j+63:i] - SRC3[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

```

**VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN

IF j \*is even\*

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] + DEST[i+63:i])

ELSE DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] - DEST[i+63:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0



**VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

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

```

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

VFMSUBADDxxxPD \_\_m512d \_\_mm512\_fmsubadd\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c);  
 VFMSUBADDxxxPD \_\_m512d \_\_mm512\_fmsubadd\_round\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c, int r);  
 VFMSUBADDxxxPD \_\_m512d \_\_mm512\_mask\_fmsubadd\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512d b, \_\_m512d c);  
 VFMSUBADDxxxPD \_\_m512d \_\_mm512\_maskz\_fmsubadd\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b, \_\_m512d c);  
 VFMSUBADDxxxPD \_\_m512d \_\_mm512\_mask3\_fmsubadd\_pd(\_\_m512d a, \_\_m512d b, \_\_m512d c, \_\_mmask8 k);  
 VFMSUBADDxxxPD \_\_m512d \_\_mm512\_mask\_fmsubadd\_round\_pd(\_\_m512d a, \_\_mmask8 k, \_\_m512d b, \_\_m512d c, int r);  
 VFMSUBADDxxxPD \_\_m512d \_\_mm512\_maskz\_fmsubadd\_round\_pd(\_\_mmask8 k, \_\_m512d a, \_\_m512d b, \_\_m512d c, int r);  
 VFMSUBADDxxxPD \_\_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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1, xmm2, 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 VFMSUBADD213PS xmm1, xmm2, 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 VFMSUBADD231PS xmm1, xmm2, 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 VFMSUBADD132PS ymm1, ymm2, 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 VFMSUBADD213PS ymm1, ymm2, 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 VFMSUBADD231PS ymm1, ymm2, 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 VFMSUBADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMSUBADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMSUBADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMSUBADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMSUBADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMSUBADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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.
EVEX.512.66.0F38.W0 97 /r VFMSUBADD132PS zmm1 {k1}{z}, 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 VFMSUBADD213PS zmm1 {k1}{z}, 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 VFMSUBADD231PS zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## 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*;
    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*;
    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*;
    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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(DEST[i+31:i]\*SRC3[i+31:i] + SRC2[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(DEST[i+31:i]\*SRC3[i+31:i] - SRC2[i+31:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] + SRC3[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[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[j+31:i] ←
                RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] + SRC3[31:0])
            ELSE
              DEST[j+31:i] ←
                RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] + SRC3[j+31:i])
          FI;
        ELSE
          IF (EVEX.b = 1)
            THEN
              DEST[j+31:i] ←
                RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] - SRC3[j+31:i])
            ELSE
              DEST[j+31:i] ←
                RoundFPControl_MXCSR(SRC2[j+31:i]*DEST[j+31:i] - SRC3[31:0])
          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

```

**VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN

IF j \*is even\*

THEN DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*SRC3[i+31:i] + DEST[i+31:i])

ELSE DEST[i+31:i] ←

RoundFPControl(SRC2[i+31:i]\*SRC3[i+31:i] - DEST[i+31:i])

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

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

```

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 __mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 __mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

```

**SIMD Floating-Point Exceptions**

Overflow, Underflow, Invalid, Precision, Denormal

**Other Exceptions**

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9A /r VFMSUB132PD xmm1, xmm2, 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 VFMSUB213PD xmm1, xmm2, 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 VFMSUB231PD xmm1, xmm2, 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 VFMSUB132PD ymm1, ymm2, 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 VFMSUB213PD ymm1, ymm2, 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 VFMSUB231PD ymm1, ymm2, 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 VFMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMSUB132PD zmm1 {k1}{z}, 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 VFMSUB213PD zmm1 {k1}{z}, 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 VFMSUB231PD zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

RoundFPControl(DEST[i+63:i]\*SRC3[i+63:i] - SRC2[i+63:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

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

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

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN

IF (EVEX.b = 1)

THEN

DEST[i+63:i] ←

RoundFPControl\_MXCSR(DEST[i+63:i]\*SRC3[63:0] - SRC2[i+63:i])

ELSE

DEST[i+63:i] ←

RoundFPControl\_MXCSR(DEST[i+63:i]\*SRC3[i+63:i] - SRC2[i+63:i])

FI;

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0



**VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*DEST[i+63:i] - SRC3[i+63:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

RoundFPControl(SRC2[i+63:i]\*SRC3[i+63:i] - DEST[i+63:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/E n	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9A /r VFMSUB132PS xmm1, xmm2, 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 VFMSUB213PS xmm1, xmm2, 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 VFMSUB231PS xmm1, xmm2, 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 VFMSUB132PS ymm1, ymm2, 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 VFMSUB213PS ymm1, ymm2, 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 VFMSUB231PS ymm1, ymm2, 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 VFMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMSUB132PS zmm1 {k1}{z}, 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 VFMSUB213PS zmm1 {k1}{z}, 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 VFMSUB231PS zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN DEST[i+31:i] ←

RoundFPControl(DEST[i+31:i]\*SRC3[i+31:i] - SRC2[i+31:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(DEST[i+31:i]\*SRC3[31:0] - SRC2[i+31:i])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(DEST[i+31:i]\*SRC3[i+31:i] - SRC2[i+31:i])

FI;

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB13PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*DEST[i+31:i] - SRC3[i+31:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VFMSUB13PS 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[i+31:i] - DEST[i+31:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

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

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

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN

IF (EVEX.b = 1)

THEN

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[31:0] - DEST[i+31:i])

ELSE

DEST[i+31:i] ←

RoundFPControl\_MXCSR(SRC2[i+31:i]\*SRC3[i+31:i] - DEST[i+31:i])

FI;

ELSE

IF \*merging-masking\* ; 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**

```

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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1, xmm2, 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 VFMSUB213SD xmm1, xmm2, 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 VFMSUB231SD xmm1, xmm2, 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.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1 {k1}{z}, 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.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm1 {k1}{z}, 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.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

**VFMSUB132SD:** Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMSUB213SD:** Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VFMSUB231SD:** Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

**VEX.128 and EVEX encoded version:** The destination operand (also first source operand) is encoded in `reg_field`. The second source operand is encoded in `VEX.vvvv/EVEX.vvvv`. The third source operand is encoded in `rm_field`. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

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

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

### Operation

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

#### VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

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

**VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[63:0] ← 0
    FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1, xmm2, 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 VFMSUB213SS xmm1, xmm2, 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 VFMSUB231SS xmm1, xmm2, 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.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1 {k1}{z}, 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.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm1 {k1}{z}, 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.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### 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_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[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_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.



## VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9C /r VFMADD132PD xmm1, xmm2, 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 VFMADD213PD xmm1, xmm2, 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 VFMADD231PD xmm1, xmm2, 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 VFMADD132PD ymm1, ymm2, 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 VFMADD213PD ymm1, ymm2, 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 VFMADD231PD ymm1, ymm2, 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 VFMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst	B	V/V	AVX512VL 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 VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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.
EVEX.512.66.0F38.W1 9C /r VFMADD132PD zmm1 {k1}{z}, 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 VFMADD213PD zmm1 {k1}{z}, 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 VFMADD231PD zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

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 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).

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 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).

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, 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

RoundFPControl(-(DEST[i+63:i]\*SRC3[i+63:i]) + SRC2[i+63:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

RoundFPControl(-(SRC2[i+63:i]\*DEST[i+63:i]) + SRC3[i+63:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN DEST[i+63:i] ←

RoundFPControl(-(SRC2[i+63:i]\*SRC3[i+63:i]) + DEST[i+63:i])

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[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**

```

VFMADDxxxPD __m512d __mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d __mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d __mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d __mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d __mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d __mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d __mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d __mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD __m256d __mm256_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d __mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d __mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d __mm_mask_fnmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d __mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d __mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d __mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9C /r VFMADD132PS xmm1, xmm2, 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 VFMADD213PS xmm1, xmm2, 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 VFMADD231PS xmm1, xmm2, 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 VFMADD132PS ymm1, ymm2, 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 VFMADD213PS ymm1, ymm2, 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 VFMADD231PS ymm1, ymm2, 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 VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512VL 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 VFMADD213PS zmm1 {k1}{z}, 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 VFMADD231PS zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## 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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[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_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
    RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+31:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+31:i] ← 0
    FI
  FI;
ENDFOR
DEST[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_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+31:i] ←
      RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[i+31:i] remains unchanged*
      ELSE                             ; zeroing-masking
        DEST[i+31:i] ← 0
      FI
    FI;
  ENDFOR
  DEST[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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9D /r VFMADD132SD xmm1, xmm2, 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 VFMADD213SD xmm1, xmm2, 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 VFMADD231SD xmm1, xmm2, 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.LIG.66.0F38.W1 9D /r VFMADD132SD xmm1 {k1}{z}, 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.LIG.66.0F38.W1 AD /r VFMADD213SD xmm1 {k1}{z}, 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.LIG.66.0F38.W1 BD /r VFMADD231SD xmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

VFMADD132SD: 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).

VFMADD213SD: 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).

VFMADD231SD: 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  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]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

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

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[63:0]  $\leftarrow$  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]  $\leftarrow$  0

FI;

FI;

DEST[127:64]  $\leftarrow$  DEST[127:64]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.



## VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9D /r VFMADD132SS xmm1, xmm2, 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 VFMADD213SS xmm1, xmm2, 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 VFMADD231SS xmm1, xmm2, 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.LIG.66.0F38.W0 9D /r VFMADD132SS xmm1 {k1}{z}, 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.LIG.66.0F38.W0 AD /r VFMADD213SS xmm1 {k1}{z}, 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.LIG.66.0F38.W0 BD /r VFMADD231SS xmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### 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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
FI;
DEST[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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
        ELSE ; zeroing-masking
            THEN DEST[31:0] ← 0
        FI;
FI;
DEST[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_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
  FI;
DEST[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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFNSUB132PD/VFNSUB213PD/VFNSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 9E /r VFNSUB132PD xmm1, xmm2, 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 VFNSUB213PD xmm1, xmm2, 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 VFNSUB231PD xmm1, xmm2, 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 VFNSUB132PD ymm1, ymm2, 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 VFNSUB213PD ymm1, ymm2, 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 VFNSUB231PD ymm1, ymm2, 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 VFNSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFNSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFNSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VFNSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFNSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VFNSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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.
EVEX.512.66.0F38.W1 9E /r VFNSUB132PD zmm1 {k1}{z}, 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 VFNSUB213PD zmm1 {k1}{z}, 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 VFNSUB231PD zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## 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_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

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

```

**VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
  FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN DEST[i+63:i] ←
      RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
        ELSE ; zeroing-masking
          DEST[i+63:i] ← 0
        FI
    FI;
  ENDFOR
  DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**  
 (KL, VL) = (2, 128), (4, 256), (8, 512)

```

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

```



**VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

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

```

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

```

VFNMSUBxxxPD __m512d __mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d __mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d __mm512_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d __mm512_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMSUBxxxPD __m256d __mm256_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d __mm256_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d __mm256_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m128d __mm_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d __mm_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d __mm_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxPD __m128d __mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNSUB132PS/VFNSUB213PS/VFNSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 9E /r VFNSUB132PS xmm1, xmm2, 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 VFNSUB213PS xmm1, xmm2, 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 VFNSUB231PS xmm1, xmm2, 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 VFNSUB132PS ymm1, ymm2, 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 VFNSUB213PS ymm1, ymm2, 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.W0 BE /r VFNSUB231PS ymm1, ymm2, 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 VFNSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFNSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFNSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VFNSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFNSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFNSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VFNSUB132PS zmm1 {k1}{z}, 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 VFNSUB213PS zmm1 {k1}{z}, 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 VFNSUB231PS zmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

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

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

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

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

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

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

## Operation

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

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

```

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

```

**VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)**

```

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

```

**VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)**

```

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

```

**VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
            ELSE ; zeroing-masking
                DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

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

```

**VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)**

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

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

```

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

```

**VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)**

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

```

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

```

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

```

VFNMSUBxxxPS __m512 __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 Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

## VFNSUB132SD/VFNSUB213SD/VFNSUB231SD—Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W1 9F /r VFNSUB132SD xmm1, xmm2, 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 VFNSUB213SD xmm1, xmm2, 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 VFNSUB231SD xmm1, xmm2, 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.LIG.66.0F38.W1 9F /r VFNSUB132SD xmm1 {k1}{z}, 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.LIG.66.0F38.W1 AF /r VFNSUB213SD xmm1 {k1}{z}, 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.LIG.66.0F38.W1 BF /r VFNSUB231SD xmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### 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).

#### VFNSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```

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

```

#### VFNSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```

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

```

**VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFNMSSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.LIG.66.0F38.W0 9F /r VFNMSSUB132SS xmm1, xmm2, 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 VFNMSSUB213SS xmm1, xmm2, 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 VFNMSSUB231SS xmm1, xmm2, 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.LIG.66.0F38.W0 9F /r VFNMSSUB132SS xmm1 {k1}{z}, 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.LIG.66.0F38.W0 AF /r VFNMSSUB213SS xmm1 {k1}{z}, 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.LIG.66.0F38.W0 BF /r VFNMSSUB231SS xmm1 {k1}{z}, 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	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

**VFNMSSUB132SS:** 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).

**VFNSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)**

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0]  $\leftarrow$  RoundFPControl(-(DEST[31:0]\*SRC3[31:0]) - SRC2[31:0])

ELSE

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

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0]  $\leftarrow$  0

FI;

FI;

DEST[127:32]  $\leftarrow$  DEST[127:32]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)**

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

THEN

SET\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

IF k1[0] or \*no writemask\*

THEN DEST[31:0]  $\leftarrow$  RoundFPControl(-(SRC2[31:0]\*DEST[31:0]) - SRC3[31:0])

ELSE

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

THEN \*DEST[31:0] remains unchanged\*

ELSE ; zeroing-masking

THEN DEST[31:0]  $\leftarrow$  0

FI;

FI;

DEST[127:32]  $\leftarrow$  DEST[127:32]

DEST[MAXVL-1:128]  $\leftarrow$  0

**VFNMSSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)**

```

IF (EVEX.b = 1) and SRC3 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
  THEN DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE ; zeroing-masking
      THEN DEST[31:0] ← 0
    FI;
FI;
DEST[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 Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

## VFPCLASSPD—Tests Types Of a Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, 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)	NA	NA

### Description

The FPCLASSPD instruction checks the packed double precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX\_KL-1:8/4/2] of the destination are cleared.

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

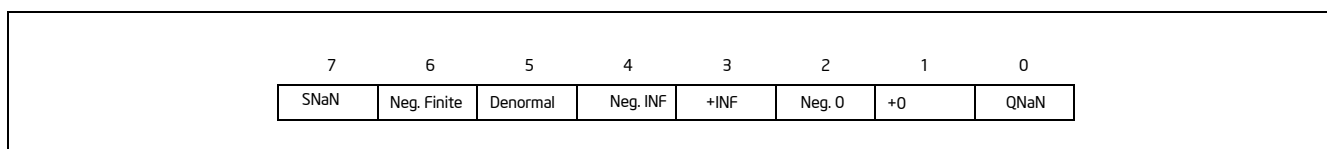


Figure 5-13. Imm8 Byte Specifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS

Table 5-4. Classifier Operations for VFPCLASSPD/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
    Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    Nlnf_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 Plnf_res ) OR
              ( imm8[4] AND Nlnf_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 Exceptions Type E4

#UD                    If EVEX.vvvv != 1111B.



## VFPCLASSPS—Tests Types Of a Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, zmm2/m512/m32bcst, imm8	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)	NA	NA

### Description

The FPCLASSPS instruction checks the packed single-precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX\_KL-1:16/8/4] of the destination are cleared.

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

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[jj] OR \*no writemask\*

THEN

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

THEN

DEST[jj] ← CheckFPClassDP(SRC1[31:0], imm8[7:0]);

ELSE

DEST[jj] ← CheckFPClassDP(SRC1[j+31:i], imm8[7:0]);

FI;

ELSE DEST[jj] ← 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 Exceptions Type E4

#UD If EVEX.vvvv != 1111B.

## VFPCLASSSD—Tests Types Of a Scalar Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEEX.LIG.66.0F3A.W1 67 /r ib VFPCLASSSD k2 {k1}, 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)	NA	NA

### 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.

EVEEX.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
Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
Nlnf_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 Plnf_res ) OR
           ( imm8[4] AND Nlnf_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 Exceptions Type E6

#UD If EVEX.vvvv != 1111B.

## VFPCLASSSS—Tests Types Of a Scalar Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEEX.LIG.66.0F3A.WO 67 /r VFPCLASSSS k2 {k1}, 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)	NA	NA

### 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.

EVEEX.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
    Plnf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    Nlnf_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 Plnf_res ) OR
              ( imm8[4] AND Nlnf_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 Exceptions Type E6  
#UD If EVEX.vvvv != 1111B.

## VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/3 2-bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 92 /r VGATHERDPD <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.128.66.0F38.W1 93 /r VGATHERQPD <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.256.66.0F38.W1 92 /r VGATHERDPD <i>ymm1</i> , <i>vm32x</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather double-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.256.66.0F38.W1 93 /r VGATHERQPD <i>ymm1</i> , <i>vm64y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather double-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg ( <i>r</i> , <i>w</i> )	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv ( <i>r</i> , <i>w</i> )	NA

### 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**

$DEST \leftarrow SRC1$ ;  
 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;  
 $MASK \leftarrow SRC3$ ;

**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[j + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[j + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[j+63:i])*SCALE + DISP);
  IF MASK[63:i] THEN
    DEST[j + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
  FI;
  MASK[j + 63:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;

```

**VGATHERQPD (VEX.256 version)**

```

MASK[MAXVL-1:256] ← 0;
FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR
DEST[MAXVL-1:256] ← 0;

```

**VGATHERDPD (VEX.256 version)**

```

MASK[MAXVL-1:256] ← 0;
FOR j ← 0 to 3
  i ← j * 64;
  IF MASK[63:i] THEN
    MASK[i + 63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 63:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  k ← j * 32;
  i ← j * 64;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
  IF MASK[63:i] THEN
    DEST[i + 63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 63:i] ← 0;
ENDFOR
DEST[MAXVL-1:256] ← 0;

```

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

VGATHERDPD: `__m128d _mm_i32gather_pd (double const * base, __m128i index, const int scale);`

VGATHERDPD: `__m128d _mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);`

VGATHERDPD: `__m256d _mm256_i32gather_pd (double const * base, __m128i index, const int scale);`

VGATHERDPD: `__m256d _mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int scale);`

VGATHERQPD: `__m128d _mm_i64gather_pd (double const * base, __m128i index, const int scale);`

VGATHERQPD: `__m128d _mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);`

VGATHERQPD: `__m256d _mm256_i64gather_pd (double const * base, __m256i index, const int scale);`

VGATHERQPD: `__m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);`

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type 12.

## VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 92 /r VGATHERDPS <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	A	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.128.66.0F38.W0 93 /r VGATHERQPS <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	A	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.256.66.0F38.W0 92 /r VGATHERDPS <i>ymm1</i> , <i>vm32y</i> , <i>ymm2</i>	A	V/V	AVX2	Using dword indices specified in <i>vm32y</i> , gather single-precision FP values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.256.66.0F38.W0 93 /r VGATHERQPS <i>xmm1</i> , <i>vm64y</i> , <i>xmm2</i>	A	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather single-precision FP values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg ( <i>r,w</i> )	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv ( <i>r, w</i> )	NA

### Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zero the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zero the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

### Operation

DEST ← SRC1;

BASE\_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK ← SRC3;

**VGATHERDPS (VEX.128 version)**

```

MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;

```

**VGATHERQPS (VEX.128 version)**

```

MASK[MAXVL-1:64] ← 0;
FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:64] ← 0;

```

**VGATHERDPS (VEX.256 version)**

```

MASK[MAXVL-1:256] ← 0;
FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 7
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:256] ← 0;

```

**VGATHERQPS (VEX.256 version)**

```

MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i +31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;

```

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

VGATHERDPS: `__m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);`

VGATHERDPS: `__m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);`

VGATHERDPS: `__m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);`

VGATHERDPS: `__m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);`

VGATHERQPS: `__m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);`

VGATHERQPS: `__m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);`

VGATHERQPS: `__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 Exceptions Type 12.



## VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 92 /vsib VGATHERDPS xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 92 /vsib VGATHERDPS ymm1 {k1}, vm32y	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 92 /vsib 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.
EVEX.128.66.0F38.W1 92 /vsib VGATHERDPD xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 92 /vsib VGATHERDPD ymm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 92 /vsib 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, VectorReg(R): VSIB:index	NA	NA

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

$i \leftarrow j * 32$

    IF  $k1[j]$

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

$MEM[BASE\_ADDR +$

$SignExtend(VINDEX[i+31:i]) * SCALE + DISP]$

$k1[j] \leftarrow 0$

    ELSE  $*DEST[i+31:i] \leftarrow$  remains unchanged\*

    FI;

ENDFOR

$k1[KL-1:0] \leftarrow 0$

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

#### VGATHERDPD (EVEX encoded version)

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

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

$i \leftarrow j * 64$

$k \leftarrow j * 32$

    IF  $k1[j]$

        THEN  $DEST[i+63:i] \leftarrow MEM[BASE\_ADDR +$

$SignExtend(VINDEX[k+31:k]) * SCALE + DISP]$

$k1[j] \leftarrow 0$

    ELSE  $*DEST[i+63:i] \leftarrow$  remains unchanged\*

    FI;

ENDFOR

$k1[KL-1:0] \leftarrow 0$

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

**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 Exceptions Type E12.

## VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x	A	V/V	AVX512VL 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 VGATHERQPS xmm1 {k1}, vm64y	A	V/V	AVX512VL 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 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 VGATHERQPD xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 93 /vsib 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, VectorReg(R): VSIB:index	NA	NA

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

$i \leftarrow j * 32$

$k \leftarrow j * 64$

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

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

$\text{MEM}[\text{BASE\_ADDR} + (\text{VINDEX}[k+63:k]) * \text{SCALE} + \text{DISP}]$

$k1[j] \leftarrow 0$

ELSE \*DEST[i+31:i] ← remains unchanged\*

FI;

ENDFOR

$k1[\text{MAX\_KL}-1:\text{KL}] \leftarrow 0$

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

#### VGATHERQPD (EVEX encoded version)

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

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 64$

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

THEN  $\text{DEST}[i+63:i] \leftarrow \text{MEM}[\text{BASE\_ADDR} + (\text{VINDEX}[i+63:i]) * \text{SCALE} + \text{DISP}]$

$k1[j] \leftarrow 0$

ELSE \*DEST[i+63:i] ← remains unchanged\*

FI;

ENDFOR

$k1[\text{MAX\_KL}-1:\text{KL}] \leftarrow 0$

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

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

```

VGATHERQPD __m512d __mm512_i64gather_pd(__m512i vdx, void * base, int scale);
VGATHERQPD __m512d __mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d __mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d __mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_i64gather_ps(__m512i vdx, void * base, int scale);
VGATHERQPS __m256 __mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 __mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 __mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E12.

## VGETEXPPD—Convert Exponents of Packed DP FP Values to DP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 42 /r VGETEXPPD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W1 42 /r VGETEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W1 42 /r VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination under writemask k1.

### Instruction Operand Encoding

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

### Description

Extracts the biased exponents from the normalized DP FP representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double-precision FP value and written to the corresponding qword elements of the destination operand (the first operand) as DP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

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

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-5.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **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 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 FP number. In case of denormal number it has the value of ZERO.
  Src.Jbit ← 0;
  Dst.exp ← 1;
  Dst.fraction ← SRC[51:0];
  WHILE(Src.Jbit = 0)
  {
    Src.Jbit ← Dst.fraction[51];      // Get the fraction MSB
    Dst.fraction ← Dst.fraction << 1; // One bit shift left
    Dst.exp--;                       // Decrement the exponent
  }
  Dst.fraction ← 0;                 // zero out fraction bits
  Dst.sign ← 1;                    // Return negative sign
  TMP[63:0] ← MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
  Return (TMP[63:0]);
}
```

```
ConvertExpDPFP(SRC[63:0])
```

```
{
  Src.sign ← 0;                    // Zero out sign bit
  Src.exp ← SRC[62:52];
  Src.fraction ← SRC[51:0];
  // Check for NaN
  IF (SRC = NaN)
  {
    IF ( SRC = SNAN ) SET IE;
    Return QNAN(SRC);
  }
  // Check for +INF
  IF (SRC = +INF) Return (SRC);

  // check if zero operand
  IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
}
ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
{
  IF ((Src.exp = 0) AND (Src.fraction != 0))
  {
    TMP[63:0] ← NormalizeExpTinyDPFP(SRC[63:0]); // Get Normalized Exponent
    Set #DE
  }
  ELSE // exponent value is correct
  {
    TMP[63:0] ← (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction);
  }
  TMP ← SAR(TMP, 52); // Shift Arithmetic Right
  TMP ← TMP - 1023; // Subtract Bias
  Return CvtI2D(TMP); // Convert INT to Double-Precision FP number
}
}
```



**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 Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 42 /r VGETEXPPS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W0 42 /r VGETEXPPS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	A	V/V	AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.

### Instruction Operand Encoding

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

### Description

Extracts the biased exponents from the normalized SP FP representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision FP value and written to the corresponding dword elements of the destination operand (the first operand) as SP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

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

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-6.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

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

Table 5-6. VGETEXPPS/SS Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	If (SRC = SNaN) then #IE If (SRC = denormal) then #DE
0 <  src1  < INF	floor(log <sub>2</sub> ( 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	0	1
Cvt_P12PS(01h) = 2 <sup>0</sup>	0	0	1	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 FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit ← 0;
    Dst.exp ← 1;
    Dst.fraction ← SRC[22:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit ← Dst.fraction[22]; // Get the fraction MSB
        Dst.fraction ← Dst.fraction << 1; // One bit shift left
        Dst.exp--; // Decrement the exponent
    }
    Dst.fraction ← 0; // zero out fraction bits
    Dst.sign ← 1; // Return negative sign
    TMP[31:0] ← MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
    Return (TMP[31:0]);
}
```

ConvertExpSPFP(SRC[31:0])

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

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
}
ELSE // check if denormal operand (notice that MXCSR.DAZ = 0)
{
```

```

    IF ((Src.exp = 0) AND (Src.fraction != 0))
    {
        TMP[31:0] ← NormalizeExpTinySPFP(SRC[31:0]);    // Get Normalized Exponent
        Set #DE
    }
    ELSE // exponent value is correct
    {
        TMP[31:0] ← (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction);
    }
    TMP ← SAR(TMP, 23); // Shift Arithmetic Right
    TMP ← TMP - 127; // Subtract Bias
    Return CvtI2D(TMP); // Convert INT to Single-Precision FP number
}
}

```

**VGETEXPPS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN

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

THEN

DEST[i+31:i] ←

ConvertExpSPFP(SRC[31:0])

ELSE

DEST[i+31:i] ←

ConvertExpSPFP(SRC[i+31:i])

FI;

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

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

```

VGETEXPPS __m512 _mm512_getexp_ps(__m512 a);
VGETEXPPS __m512 _mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_maskz_getexp_ps(__mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_getexp_round_ps(__m512 a, int sae);
VGETEXPPS __m512 _mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae);
VGETEXPPS __m512 _mm512_maskz_getexp_round_ps(__mmask16 k, __m512 a, int sae);
VGETEXPPS __m256 _mm256_getexp_ps(__m256 a);
VGETEXPPS __m256 _mm256_mask_getexp_ps(__m256 s, __mmask8 k, __m256 a);
VGETEXPPS __m256 _mm256_maskz_getexp_ps(__mmask8 k, __m256 a);
VGETEXPPS __m128 _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 Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.

## VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W1 43 /r VGETEXPSD xmm1 {k1}{z}, 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 DP FP value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.

### 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)	NA

### Description

Extracts the biased exponent from the normalized DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location. The low quadword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-5.

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 Exceptions Type E3.

## VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}{z}, xmm2, 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 SP FP 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)	NA

### Description

Extracts the biased exponent from the normalized SP FP representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision FP value and written to the destination operand (the first operand) as SP FP numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location. The the low doubleword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-6.

The formula is:

$$\text{GETEXP}(x) = \text{floor}(\log_2(|x|))$$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

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

### Operation

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

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

#### VGETEXPSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
  THEN DEST[31:0] ←
    ConvertExpDPFP(SRC2[31:0])
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[31:0] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[31:0] ← 0
    FI
  FI;
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 Exceptions Type E3.

## VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 26 /r ib VGETMANTPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W1 26 /r ib VGETMANTPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using <i>imm8</i> for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F	Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using <i>imm8</i> 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	NA

### Description

Convert double-precision floating values in the source operand (the second operand) to DP FP values with the mantissa normalization and sign control specified by the *imm8* byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask *k1*. The normalized mantissa is specified by *interv* (*imm8*[1:0]) and the sign control (*sc*) is specified by bits 3:2 of the immediate byte.

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

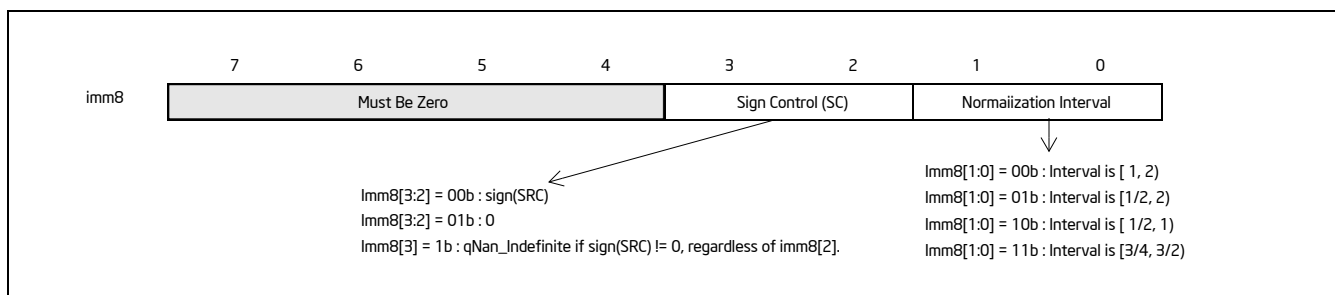


Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input DP FP value *x*, The conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

where:

$$1 \leq |x.significand| < 2$$

Unbiased exponent *k* depends on the interval range defined by *interv* and whether the exponent of the source is even or odd. The sign of the final result is determined by *sc* and the source sign.

If  $interv \neq 0$  then  $k = -1$ , otherwise  $k = 0$ . The encoded value of  $imm8[1:0]$  and sign control are shown in Figure 5-15.

Each converted DP FP result is encoded according to the sign control, the unbiased exponent  $k$  (adding bias) and a mantissa normalized to the range specified by  $interv$ .

The `GetMant()` function follows Table 5-7 when dealing with floating-point special numbers.

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

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

**Table 5-7. GetMant() Special Float Values Behavior**

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore <i>interv</i> 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 ELSE -1.0	Ignore <i>interv</i>
$-\infty$	IF (SC[1]) THEN {QNaN_Indefinite} ELSE { IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore <i>interv</i> If (SC[1]) then #IE
negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)	If (SC[1]) then #IE

### Operation

```
GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[63];           // Get sign bit
    Dst.exp ← SRC[62:52]; ; Get original exponent value
    Dst.fraction ← SRC[51:0];; Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 07FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 07FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {
        IF (SRC = SNaN) {Set #IE;}
        Return QNaN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand))
    {
        Dst.exp ← 03FFh;           // Override exponent with BIAS
        Return ((Dst.sign<<63) | (Dst.exp<<52) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[63] = 1) AND SignCtrl[1])
    {
        Set #IE;
        Return QNaN_Indefinite;
    }
}
```

```

// Checking for denormal operands
IF (DenormOperand)
{
  IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
  ELSE
  {
    // Jbit is the hidden integral bit. Zero in case of denormal operand.
    Src.Jbit ← 0; // Zero Src Jbit
    Dst.exp ← 03FFh; // Override exponent with BIAS
    WHILE (Src.Jbit = 0) { // normalize mantissa
      Src.Jbit ← Dst.fraction[51]; // Get the fraction MSB
      Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
      Dst.exp--; // Adjust the exponent
    }
    SET #DE; // Set DE bit
  }
}
// At this point, Dst.fraction is normalized.
// Checking for exponent response
Unbiased.exp ← Dst.exp - 03FFh; // subtract the bias from exponent
IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
SignalingBit ← Dst.fraction[51];
CASE (interv[1:0])
  00: Dst.exp ← 03FFh; // This is the bias
  01: Dst.exp ← (IsOddExp) ? 03FEh : 03FFh; // either bias-1, or bias
  10: Dst.exp ← 03FEh; // bias-1
  11: Dst.exp ← (SignalingBit) ? 03FEh : 03FFh; // either bias-1, or bias
ESAC
// At this point Dst.exp has the correct result. Form the final destination
DEST[63:0] ← (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
Return (DEST);
}

```

**VGETMANTPD (EVEX encoded versions)**

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

SignCtrl[1:0] ← IMM8[3:2];

Interv[1:0] ← IMM8[1:0];

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] ← GetNormalizedMantissaDP(SRC[63:0], SignCtrl, Interv)

ELSE

DEST[i+63:i] ← GetNormalizedMantissaDP(SRC[j+63:i], SignCtrl, Interv)

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**

```

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 Exceptions Type E2.

#UD                    If EVEX.vvvv != 1111B.

## VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 26 /r ib VGETMANTPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector xmm2/m128/m32bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W0 26 /r ib VGETMANTPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector ymm2/m256/m32bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W0 26 /r ib VGETMANTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F	Get normalized mantissa from float32 vector zmm2/m512/m32bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

### 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	NA

### Description

Convert single-precision floating values in the source operand (the second operand) to SP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input SP FP value  $x$ , The conversion operation is:

$$GetMant(x) = \pm 2^k |x.significand|$$

where:

$$1 \leq |x.significand| < 2$$

Unbiased exponent  $k$  depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then  $k = -1$ , otherwise  $k = 0$ . The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted SP FP result is encoded according to the sign control, the unbiased exponent  $k$  (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-7 when dealing with floating-point special numbers.

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

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

**Operation**

```

GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[31];           // Get sign bit
    Dst.exp ← SRC[30:23]; // Get original exponent value
    Dst.fraction ← SRC[22:0]; // Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 0FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 0FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {
        IF (SRC = SNaN) {Set #IE;}
        Return QNaN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand))
    {
        Dst.exp ← 07Fh; // Override exponent with BIAS
        Return ((Dst.sign << 31) | (Dst.exp << 23) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[31] = 1) AND SignCtrl[1])
    {
        Set #IE;
        Return QNaN_Indefinite;
    }
    // Checking for denormal operands
    IF (DenormOperand)
    {
        IF (MXCSR.DAZ=1) Dst.fraction ← 0; // Zero out fraction
        ELSE
        {
            // Jbit is the hidden integral bit. Zero in case of denormal operand.
            Src.Jbit ← 0; // Zero Src Jbit
            Dst.exp ← 07Fh; // Override exponent with BIAS
            WHILE (Src.Jbit = 0) { // normalize mantissa
                Src.Jbit ← Dst.fraction[22]; // Get the fraction MSB
                Dst.fraction ← (Dst.fraction << 1); // Start normalizing the mantissa
                Dst.exp--; // Adjust the exponent
            }
            SET #DE; // Set DE bit
        }
    }
    // At this point, Dst.fraction is normalized.
    // Checking for exponent response
    Unbiased.exp ← Dst.exp - 07Fh; // subtract the bias from exponent
    IsOddExp ← Unbiased.exp[0]; // recognized unbiased ODD exponent
    SignalingBit ← Dst.fraction[22];
    CASE (interv[1:0])
        00: Dst.exp ← 07Fh; // This is the bias
        01: Dst.exp ← (IsOddExp) ? 07Eh : 07Fh; // either bias-1, or bias
        10: Dst.exp ← 07Eh; // bias-1
        11: Dst.exp ← (SignalingBit) ? 07Eh : 07Fh; // either bias-1, or bias
    ESAC

    // Form the final destination
    DEST[31:0] ← (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
}

```

```

    Return (DEST);
}

```

**VGETMANTPS (EVEX encoded versions)**

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

SignCtrl[1:0] ← IMM8[3:2];

Interv[1:0] ← IMM8[1:0];

FOR j ← 0 TO KL-1

i ← j \* 32

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

THEN

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

THEN

DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[31:0], SignCtrl, Interv)

ELSE

DEST[i+31:i] ← GetNormalizedMantissaSP(SRC[j+31:i], SignCtrl, Interv)

FI;

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

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

VGETMANTPS \_\_m512 \_\_mm512\_getmant\_ps( \_\_m512 a, enum intv, enum sgn);

VGETMANTPS \_\_m512 \_\_mm512\_mask\_getmant\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, enum intv, enum sgn);

VGETMANTPS \_\_m512 \_\_mm512\_maskz\_getmant\_ps(\_\_mmask16 k, \_\_m512 a, enum intv, enum sgn);

VGETMANTPS \_\_m512 \_\_mm512\_getmant\_round\_ps( \_\_m512 a, enum intv, enum sgn, int r);

VGETMANTPS \_\_m512 \_\_mm512\_mask\_getmant\_round\_ps(\_\_m512 s, \_\_mmask16 k, \_\_m512 a, enum intv, enum sgn, int r);

VGETMANTPS \_\_m512 \_\_mm512\_maskz\_getmant\_round\_ps(\_\_mmask16 k, \_\_m512 a, enum intv, enum sgn, int r);

VGETMANTPS \_\_m256 \_\_mm256\_getmant\_ps( \_\_m256 a, enum intv, enum sgn);

VGETMANTPS \_\_m256 \_\_mm256\_mask\_getmant\_ps(\_\_m256 s, \_\_mmask8 k, \_\_m256 a, enum intv, enum sgn);

VGETMANTPS \_\_m256 \_\_mm256\_maskz\_getmant\_ps( \_\_mmask8 k, \_\_m256 a, enum intv, enum sgn);

VGETMANTPS \_\_m128 \_\_mm\_getmant\_ps( \_\_m128 a, enum intv, enum sgn);

VGETMANTPS \_\_m128 \_\_mm\_mask\_getmant\_ps(\_\_m128 s, \_\_mmask8 k, \_\_m128 a, enum intv, enum sgn);

VGETMANTPS \_\_m128 \_\_mm\_maskz\_getmant\_ps( \_\_mmask8 k, \_\_m128 a, enum intv, enum sgn);

**SIMD Floating-Point Exceptions**

Denormal, Invalid

**Other Exceptions**

See Exceptions Type E2.

#UD If EVEX.vvvv != 1111B.



## VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEEX.LIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F	Extract the normalized mantissa of the low float64 element in xmm3/m64 using <i>imm8</i> 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)	NA

### Description

Convert the double-precision floating values in the low quadword element of the second source operand (the third operand) to DP FP value with the mantissa normalization and sign control specified by the *imm8* byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by *interv* (*imm8*[1:0]) and the sign control (*sc*) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

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

where:

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

Unbiased exponent *k* depends on the interval range defined by *interv* and whether the exponent of the source is even or odd. The sign of the final result is determined by *sc* and the source sign.

If *interv* != 0 then *k* = -1, otherwise *k* = 0. The encoded value of *imm8*[1:0] and sign control are shown in Figure 5-15.

The converted DP FP result is encoded according to the sign control, the unbiased exponent *k* (adding bias) and a mantissa normalized to the range specified by *interv*.

The *GetMant*() function follows Table 5-7 when dealing with floating-point special numbers.

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

**Operation**

// GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

**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] ←
    GetNormalizedMantissaDP(SRC2[63:0], SignCtrl, Interv)
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE                             ; zeroing-masking
      DEST[63:0] ← 0
    FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[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 Exceptions Type E3.

## VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 {k1}{z}, xmm2, 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)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

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

The conversion operation is:

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

where:

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

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-7 when dealing with floating-point special numbers.

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

**Operation**

// GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

**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] ←
    GetNormalizedMantissaSP(SRC2[31:0], SignCtrl, Interv)
  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[127:64]
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 Exceptions Type E3.

## VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, 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 VINSERTF32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	AVX512VL 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 VINSERTF32X4 zmm1 {k1}{z}, zmm2, 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 VINSERTF64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	AVX512VL 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 VINSERTF64X2 zmm1 {k1}{z}, zmm2, 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 VINSERTF32X8 zmm1 {k1}{z}, zmm2, 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 VINSERTF64X4 zmm1 {k1}{z}, zmm2, 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	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv	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 Exceptions Type 6; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.



## VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 38 /r ib VINSERTI128 ymm1, ymm2, 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 VINSERTI32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	AVX512VL 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 VINSERTI32X4 zmm1 {k1}{z}, zmm2, 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 VINSERTI64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	AVX512VL 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 VINSERTI64X2 zmm1 {k1}{z}, zmm2, 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 VINSERTI32X8 zmm1 {k1}{z}, zmm2, 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 VINSERTI64X4 zmm1 {k1}{z}, zmm2, 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	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv	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 Exceptions Type 6; additionally

#UD If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

## VMASKMOV—Conditional SIMD Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 2C /r VMASKMOVPS <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX	Conditionally load packed single-precision values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.256.66.0F38.W0 2C /r VMASKMOVPS <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX	Conditionally load packed single-precision values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.128.66.0F38.W0 2D /r VMASKMOVPD <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX	Conditionally load packed double-precision values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.256.66.0F38.W0 2D /r VMASKMOVPD <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX	Conditionally load packed double-precision values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.128.66.0F38.W0 2E /r VMASKMOVPS <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX	Conditionally store packed single-precision values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.256.66.0F38.W0 2E /r VMASKMOVPS <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX	Conditionally store packed single-precision values from <i>ymm2</i> using mask in <i>ymm1</i> .
VEX.128.66.0F38.W0 2F /r VMASKMOVPD <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX	Conditionally store packed double-precision values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.256.66.0F38.W0 2F /r VMASKMOVPD <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX	Conditionally store packed double-precision values from <i>ymm2</i> using mask in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

### Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm\_field, and the destination register is encoded in reg\_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg\_field, and the destination memory location is encoded in rm\_field.

## Operation

### VMASKMOVPS - 128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← 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 Exceptions Type 6 (No AC# reported for any mask bit combinations); additionally

#UD                    If VEX.W = 1.

## VPBLEND – Blend Packed Dwords

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

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	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 word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

### Operation

#### 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 Exceptions Type 4; additionally

#UD If VEX.W = 1.

## VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 66 /r VPBLENDMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL 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 VPBLENDMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL 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 VPBLENDMB zmm1 {k1}{z}, 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 VPBLENDMW xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL 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 VPBLENDMW ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL 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 VPBLENDMW zmm1 {k1}{z}, 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	ModRM:r/m (r)	NA

### 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 Exceptions Type E4.

## VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 64 /r VPBLENDMD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL 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 VPBLENDMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL 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 VPBLENDMD zmm1 {k1}{z}, zmm2, 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 VPBLENDMQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL 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 VPBLENDMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL 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 VPBLENDMQ zmm1 {k1}{z}, zmm2, 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	ModRM:r/m (r)	NA

### 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 Exceptions Type E4.

## VPBROADCASTB/W/D/Q—Load with Broadcast Integer Data from General Purpose Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 7A /r VPBROADCASTB xmm1 {k1}{z}, reg	A	V/V	AVX512VL 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 VPBROADCASTB ymm1 {k1}{z}, reg	A	V/V	AVX512VL 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 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 VPBROADCASTW xmm1 {k1}{z}, reg	A	V/V	AVX512VL 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 VPBROADCASTW ymm1 {k1}{z}, reg	A	V/V	AVX512VL 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 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 VPBROADCASTD xmm1 {k1}{z}, r32	A	V/V	AVX512VL AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 {k1}{z}, r32	A	V/V	AVX512VL AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32	A	V/V	AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 {k1}{z}, r64	A	V/N.E. <sup>1</sup>	AVX512VL AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 {k1}{z}, r64	A	V/N.E. <sup>1</sup>	AVX512VL AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64	A	V/N.E. <sup>1</sup>	AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 512-bit destination subject to writemask k1.

### NOTES:

1. EVEX.W in non-64 bit is ignored; the instructions 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)	NA	NA

### 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 Exceptions Type E7NM.

#UD If EVEX.vvvv != 1111B.

## VPBROADCAST—Load Integer and Broadcast

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 78 /r 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 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 VPBROADCASTB xmm1{k1}{z}, xmm2/m8	B	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1{k1}{z}, xmm2/m8	B	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 78 /r 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 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 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 VPBROADCASTW xmm1{k1}{z}, xmm2/m16	B	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1{k1}{z}, xmm2/m16	B	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 79 /r 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 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 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 VPBROADCASTD xmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 58 /r 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 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 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 VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 59 /r VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W1 59 /r 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.
EVEX.128.66.0F38.W0 59 /r VPBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W0 59 /r VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 59 /r 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 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 VBROADCASTI32X4 ymm1 {k1}{z}, m128	D	V/V	AVX512VL 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 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 VBROADCASTI64X2 ymm1 {k1}{z}, m128	C	V/V	AVX512VL 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 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 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 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	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

#### 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 VPBROADCASTI32X4 and VPBROADCASTI64X4 are reserved and will #UD.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.  
 If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

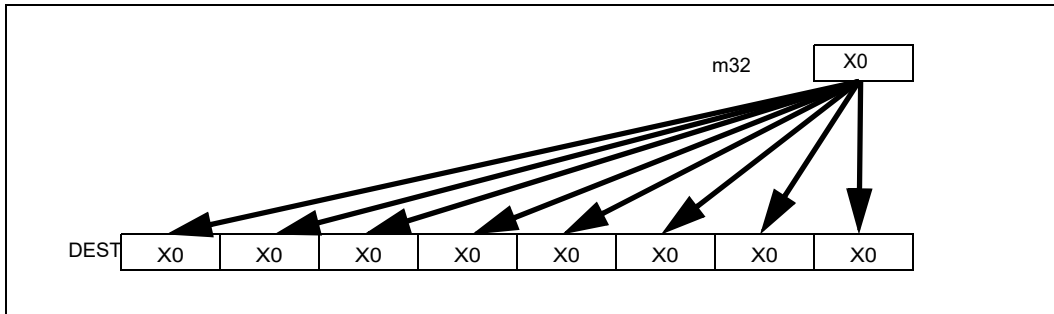


Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

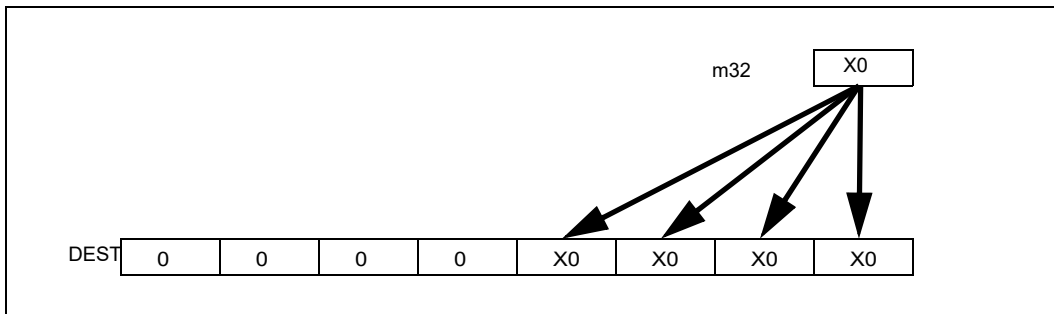


Figure 5-17. VPBROADCASTD Operation (128-bit version)

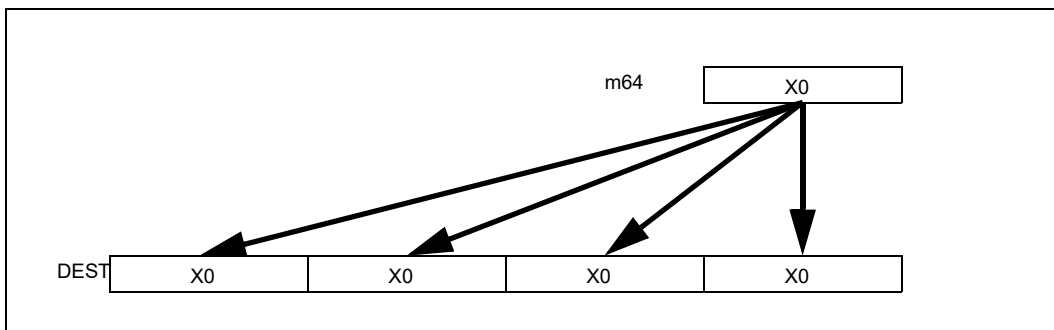


Figure 5-18. VPBROADCASTQ Operation (256-bit version)

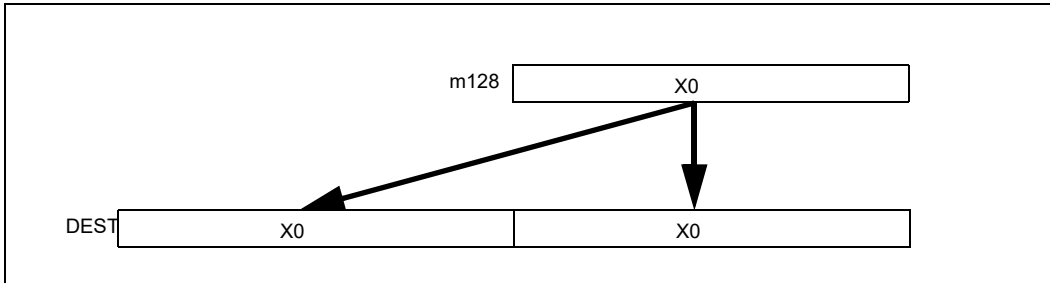


Figure 5-19. VBROADCASTI128 Operation (256-bit version)

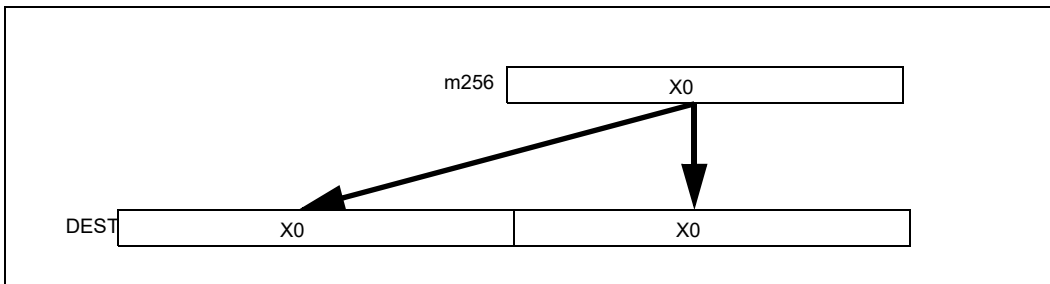


Figure 5-20. VBROADCASTI256 Operation (512-bit version)

**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 Exceptions Type 6;

EVEX-encoded instructions, syntax with reg/mem operand, see Exceptions Type E6.

#UD                    If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.  
                           If EVEX.L'L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.  
                           If EVEX.L'L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.

## VPBROADCASTM—Broadcast Mask to Vector Register

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

### 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 Exceptions Type E6NF.

## VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 3F /r ib  VPCMPB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL 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  VPCMPB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL 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 VPCMPB k1 {k2}, zmm2, 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  VPCMPUB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL 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  VPCMPUB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL 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 VPCMPUB k1 {k2}, zmm2, 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)	vvvv (r)	ModRM:r/m (r)	NA

#### Description

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

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.

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

**Table 5-8. Pseudo-Op and VPCMP\* Implementation**

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[i+7:i] OP SRC2[i+7:i];

      IF CMP = TRUE

        THEN DEST[j] ← 1;

        ELSE DEST[j] ← 0; FI;

    ELSE DEST[j] = 0 ; zeroing-masking onlyFI;

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUB (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 8

IF k2[j] OR \*no writemask\*

THEN

CMP ← SRC1[i+7:i] OP SRC2[i+7:i];

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] = 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

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

VPCMPB \_\_mmask64\_mm512\_cmp\_epi8\_mask( \_\_m512i a, \_\_m512i b, int cmp);

VPCMPB \_\_mmask64\_mm512\_mask\_cmp\_epi8\_mask( \_\_mmask64 m, \_\_m512i a, \_\_m512i b, int cmp);

VPCMPB \_\_mmask32\_mm256\_cmp\_epi8\_mask( \_\_m256i a, \_\_m256i b, int cmp);

VPCMPB \_\_mmask32\_mm256\_mask\_cmp\_epi8\_mask( \_\_mmask32 m, \_\_m256i a, \_\_m256i b, int cmp);

VPCMPB \_\_mmask16\_mm\_cmp\_epi8\_mask( \_\_m128i a, \_\_m128i b, int cmp);

VPCMPB \_\_mmask16\_mm\_mask\_cmp\_epi8\_mask( \_\_mmask16 m, \_\_m128i a, \_\_m128i b, int cmp);

VPCMPB \_\_mmask64\_mm512\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_m512i a, \_\_m512i b);

VPCMPB \_\_mmask64\_mm512\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_mmask64 m, \_\_m512i a, \_\_m512i b);

VPCMPB \_\_mmask32\_mm256\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_m256i a, \_\_m256i b);

VPCMPB \_\_mmask32\_mm256\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_mmask32 m, \_\_m256i a, \_\_m256i b);

VPCMPB \_\_mmask16\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_m128i a, \_\_m128i b);

VPCMPB \_\_mmask16\_mm\_mask\_cmp[eq|ge|gt|le|lt|neq]\_epi8\_mask( \_\_mmask16 m, \_\_m128i a, \_\_m128i b);

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\_mm\_cmp\_epu8\_mask( \_\_m128i a, \_\_m128i b, int cmp);

VPCMPUB \_\_mmask16\_mm\_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\_mm\_cmp[eq|ge|gt|le|lt|neq]\_epu8\_mask( \_\_m128i a, \_\_m128i b, int cmp);

VPCMPUB \_\_mmask16\_mm\_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.

## VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL 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 VPCMPD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL 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 VPCMPD k1 {k2}, zmm2, 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 VPCMPUD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL 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 VPCMPUD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL 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 VPCMPUD k1 {k2}, zmm2, 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 5-8.



**Operation**

CASE (COMPARISON PREDICATE) OF

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

ESAC;

**VPCMPD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k2[j] OR \*no writemask\*

THEN

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

THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF k2[j] OR \*no writemask\*

THEN

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

THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];

ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking onlyFI;

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

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

```

VPCMPD __mmask16 _mm512_cmp_epi32_mask( __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask( __m512i a, __m512i b);
VPCMPD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_cmp_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 Exceptions Type E4.

## VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL 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 VPCMPQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL 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 VPCMPQ k1 {k2}, zmm2, 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 VPCMPUQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL 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 VPCMPUQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL 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 VPCMPUQ k1 {k2}, zmm2, 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 5-8.

**Operation**

CASE (COMPARISON PREDICATE) OF

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

ESAC;

**VPCMPQ (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

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

THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];

ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPCMPUQ (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF k2[j] OR \*no writemask\*

THEN

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

THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];

ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];

FI;

IF CMP = TRUE

THEN DEST[j] ← 1;

ELSE DEST[j] ← 0; FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**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 Exceptions Type E4.

## VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 3F /r ib  VPCMPW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL 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  VPCMPW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL 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 VPCMPW k1 {k2}, zmm2, 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  VPCMPUW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL 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  VPCMPUW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL 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.
VPCMPUW k1 {k2}, zmm2, 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)	vvvv (r)	ModRM:r/m (r)	NA

### Description

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

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

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

**Operation**

CASE (COMPARISON PREDICATE) OF

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

ESAC;

**VPCMPW (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 16

IF k2[j] OR \*no writemask\*

THEN

ICMP ← SRC1[i+15:i] OP SRC2[i+15:i];

IF ICMP = 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.



## VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8B /r VPCOMPRESSD xmm1/m128 {k1}{z}, xmm2	A	V/V	AVX512VL AVX512F	Compress packed doubleword integer values from xmm2 to xmm1/m128 using controlmask k1.
EVEX.256.66.0F38.W0 8B /r VPCOMPRESSD ymm1/m256 {k1}{z}, ymm2	A	V/V	AVX512VL AVX512F	Compress packed doubleword integer values from ymm2 to ymm1/m256 using controlmask k1.
EVEX.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F	Compress packed doubleword integer values from zmm2 to zmm1/m512 using controlmask 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)	NA	NA

### 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.

## VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register

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

### Instruction Operand Encoding

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

### 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.

## VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/ Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 C4 /r VPCONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 C4 /r VPCONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD	Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 C4 /r VPCONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 C4 /r VPCONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD	Detect duplicate quad-word values in 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)	NA	NA

### 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[j+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 Exceptions Type E4NF.

### VPERM2F128 – Permute Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 06 /r ib VPERM2F128 <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX	Permute 128-bit floating-point fields in <i>ymm2</i> and <i>ymm3/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

#### 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.

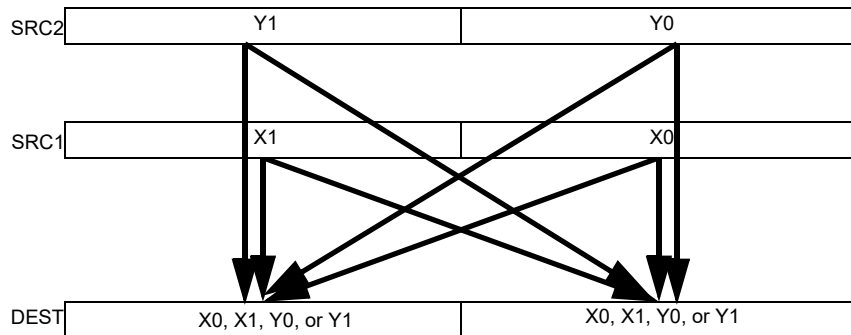


Figure 5-21. VPERM2F128 Operation

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 Exceptions Type 6; additionally

#UD                    If VEX.L = 0  
                           If VEX.W = 1.

## VPERM2I128 – Permute Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 46 /r ib VPERM2I128 <i>ymm1, ymm2, ymm3/m256, imm8</i>	RVMI	V/V	AVX2	Permute 128-bit integer data in <i>ymm2</i> and <i>ymm3/mem</i> using controls from <i>imm8</i> and store result in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	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.

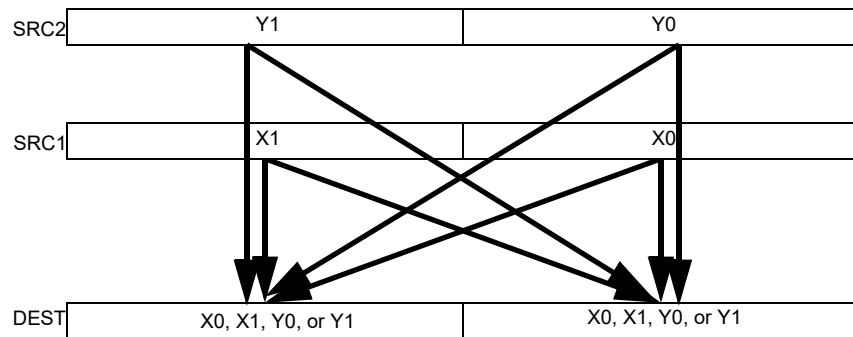


Figure 5-22. VPERM2I128 Operation

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 Exceptions Type 6; additionally

#UD                    If VEX.L = 0,  
                          If VEX.W = 1.

## VPERMB—Permute Packed Bytes Elements

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8D /r VPERMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL 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 VPERMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL 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 VPERMB zmm1 {k1}{z}, zmm2, 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)	NA

### Description

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

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

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

### Operation

#### VPERMB (EVEX encoded versions)

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

IF VL = 128:

n ← 3;

ELSE IF VL = 256:

n ← 4;

ELSE IF VL = 512:

n ← 5;

FI;

FOR j ← 0 TO KL-1:

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

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

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

ELSE IF zeroing-masking THEN

DEST[j\*8 + 7 : j\*8] ← 0;

\*ELSE

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

FI

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

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

VPERMB \_\_m512i \_\_mm512\_permutexvar\_epi8(\_\_m512i idx, \_\_m512i a);

VPERMB \_\_m512i \_\_mm512\_mask\_permutexvar\_epi8(\_\_m512i s, \_\_mmask64 k, \_\_m512i idx, \_\_m512i a);  
 VPERMB \_\_m512i \_\_mm512\_maskz\_permutexvar\_epi8(\_\_mmask64 k, \_\_m512i idx, \_\_m512i a);  
 VPERMB \_\_m256i \_\_mm256\_permutexvar\_epi8(\_\_m256i idx, \_\_m256i a);  
 VPERMB \_\_m256i \_\_mm256\_mask\_permutexvar\_epi8(\_\_m256i s, \_\_mmask32 k, \_\_m256i idx, \_\_m256i a);  
 VPERMB \_\_m256i \_\_mm256\_maskz\_permutexvar\_epi8(\_\_mmask32 k, \_\_m256i idx, \_\_m256i a);  
 VPERMB \_\_m128i \_\_mm\_permutexvar\_epi8(\_\_m128i idx, \_\_m128i a);  
 VPERMB \_\_m128i \_\_mm\_mask\_permutexvar\_epi8(\_\_m128i s, \_\_mmask16 k, \_\_m128i idx, \_\_m128i a);  
 VPERMB \_\_m128i \_\_mm\_maskz\_permutexvar\_epi8(\_\_mmask16 k, \_\_m128i idx, \_\_m128i a);

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type E4NF.nb.

## VPERMD/VPERMW—Permute Packed Doublewords/Words Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 36 /r 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 VPERMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VPERMD zmm1 {k1}{z}, zmm2, 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 VPERMW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL 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 VPERMW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL 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 VPERMW zmm1 {k1}{z}, zmm2, 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	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
B	Full	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
C	Full Mem	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

### Description

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

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

**Operation****VPERMD (EVEX encoded versions)**

```

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

```

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

```

DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[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

```

**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
  FI;
  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 Exceptions Type 4.

EVEX-encoded VPERMD, see Exceptions Type E4NF.

EVEX-encoded VPERMW, see Exceptions Type E4NF.nb.

#UD                    If VEX.L = 0.  
                           If EVEX.L'L = 0 for VPERMD.



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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 75 /r VPERMI2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512_VBMI	Permute bytes in xmm3/m128 and xmm2 using byte indexes in xmm1 and store the byte results in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 75 /r VPERMI2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512_VBMI	Permute bytes in ymm3/m256 and ymm2 using byte indexes in ymm1 and store the byte results in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 75 /r VPERMI2B zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512_VBMI	Permute bytes in zmm3/m512 and zmm2 using byte indexes in zmm1 and store the byte results in zmm1 using writemask k1.

### Instruction Operand Encoding

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

### Description

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

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

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

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

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

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

IF VL = 128:

id ← 3;

ELSE IF VL = 256:

id ← 4;

ELSE IF VL = 512:

id ← 5;

FI;

TMP\_DEST[VL-1:0] ← DEST[VL-1:0];

FOR j ← 0 TO KL-1

off ← 8\*SRC1[j\*8 + id:j\*8];

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

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

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7:j\*8] ← 0;

\*ELSE

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

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

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

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

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

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

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

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

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

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

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

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

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.

## VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 75 /r VPERMI2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL 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 VPERMI2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL 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 VPERMI2W zmm1 {k1}{z}, zmm2, 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 VPERMI2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VPERMI2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VPERMI2D zmm1 {k1}{z}, zmm2, 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 VPERMI2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VPERMI2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VPERMI2Q zmm1 {k1}{z}, zmm2, 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 VPERMI2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 77 /r VPERMI2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 77 /r VPERMI2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 77 /r VPERMI2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 77 /r VPERMI2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

### 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)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

### Description

Permutes 16-bit/32-bit/64-bit values in the 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[j+31:i] remains unchanged*
        ELSE                             ; zeroing-masking
            DEST[j+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[j+id+1] ? SRC2[63:0]

: SRC1[off+63:off]

ELSE

DEST[i+63:i] ← TMP\_DEST[j+id+1] ? SRC2[off+63:off]

: SRC1[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

```

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 Exceptions Type E4NF.

VPERMI2W: See Exceptions Type E4NF.nb.



## VPERMILPD—Permute In-Lane of Pairs of Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0D /r 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 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 VPERMILPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL 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 VPERMILPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL 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 VPERMILPD zmm1 {k1}{z}, zmm2, 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 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 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 VPERMILPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	D	V/V	AVX512VL 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 VPERMILPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	D	V/V	AVX512VL 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 VPERMILPD zmm1 {k1}{z}, 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	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

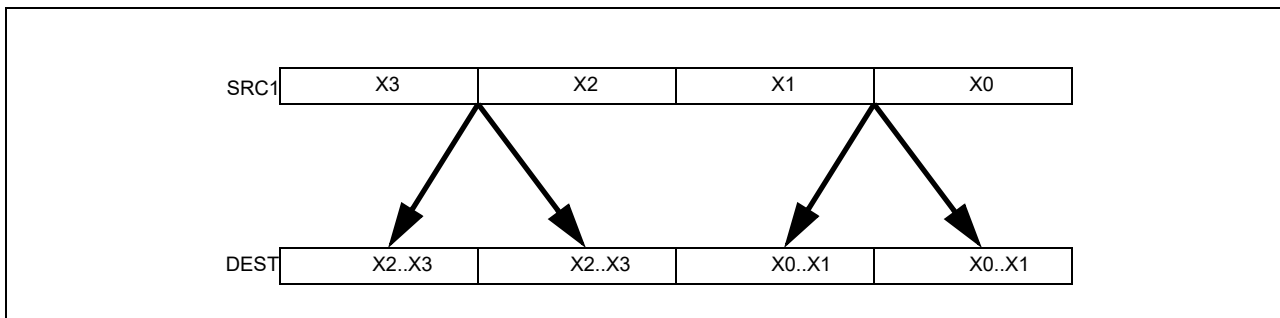
**Description**

(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).

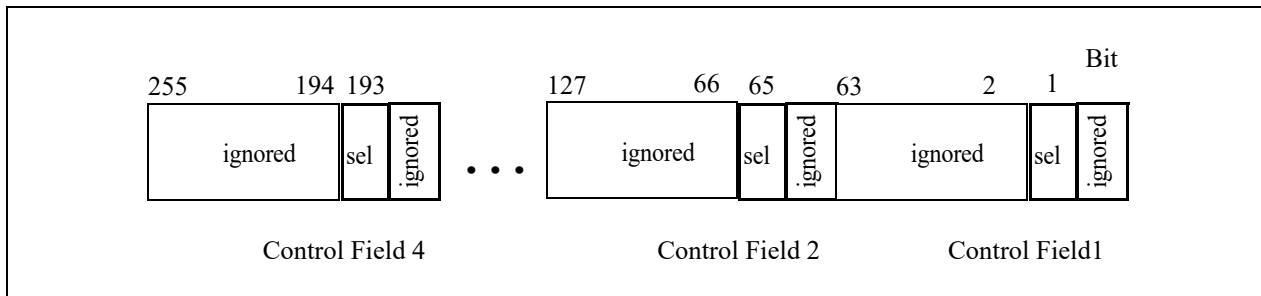
The control bits are located at bit 0 of each quadword element (see Figure 5-24). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

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.

**Operation****VPERMILPD (EVEX immediate versions)**

```

(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+63:i] ← SRC1[63:0];
        ELSE TMP_SRC1[i+63:i] ← SRC1[i+63:i];
    FI;
ENDFOR;
IF (imm8[0] = 0) THEN TMP_DEST[63:0] ← SRC1[63:0]; FI;
IF (imm8[0] = 1) THEN TMP_DEST[63:0] ← TMP_SRC1[127:64]; FI;
IF (imm8[1] = 0) THEN TMP_DEST[127:64] ← TMP_SRC1[63:0]; FI;
IF (imm8[1] = 1) THEN TMP_DEST[127:64] ← TMP_SRC1[127:64]; FI;
IF VL >= 256
    IF (imm8[2] = 0) THEN TMP_DEST[191:128] ← TMP_SRC1[191:128]; FI;
    IF (imm8[2] = 1) THEN TMP_DEST[191:128] ← TMP_SRC1[255:192]; FI;
    IF (imm8[3] = 0) THEN TMP_DEST[255:192] ← TMP_SRC1[191:128]; FI;
    IF (imm8[3] = 1) THEN TMP_DEST[255:192] ← TMP_SRC1[255:192]; FI;
FI;
IF VL >= 512
    IF (imm8[4] = 0) THEN TMP_DEST[319:256] ← TMP_SRC1[319:256]; FI;
    IF (imm8[4] = 1) THEN TMP_DEST[319:256] ← TMP_SRC1[383:320]; FI;
    IF (imm8[5] = 0) THEN TMP_DEST[383:320] ← TMP_SRC1[319:256]; FI;
    IF (imm8[5] = 1) THEN TMP_DEST[383:320] ← TMP_SRC1[383:320]; FI;
    IF (imm8[6] = 0) THEN TMP_DEST[447:384] ← TMP_SRC1[447:384]; FI;
    IF (imm8[6] = 1) THEN TMP_DEST[447:384] ← TMP_SRC1[511:448]; FI;
    IF (imm8[7] = 0) THEN TMP_DEST[511:448] ← TMP_SRC1[447:384]; FI;
    IF (imm8[7] = 1) THEN TMP_DEST[511:448] ← TMP_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
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

```

**VPERMILPD (256-bit immediate version)**

```

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64] ← SRC1[127:64]
IF (imm8[2] = 0) THEN DEST[191:128] ← SRC1[191:128]
IF (imm8[2] = 1) THEN DEST[191:128] ← SRC1[255:192]
IF (imm8[3] = 0) THEN DEST[255:192] ← SRC1[191:128]
IF (imm8[3] = 1) THEN DEST[255:192] ← SRC1[255:192]
DEST[MAXVL-1:256] ← 0

```

**VPERMILPD (128-bit immediate version)**

```

IF (imm8[0] = 0) THEN DEST[63:0] ← SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0] ← SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64] ← SRC1[63:0]
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 Exceptions Type 4; additionally

#UD If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If either (E)VEX.vvvv != 1111B and with imm8.

## VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0C /r 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 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 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 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 VPERMILPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL 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 VPERMILPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL 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 VPERMILPS zmm1 {k1}{z}, zmm2, 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 VPERMILPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	D	V/V	AVX512VL 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 VPERMILPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	D	V/V	AVX512VL 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 ibVPERMILPS zmm1 {k1}{z}, 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	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

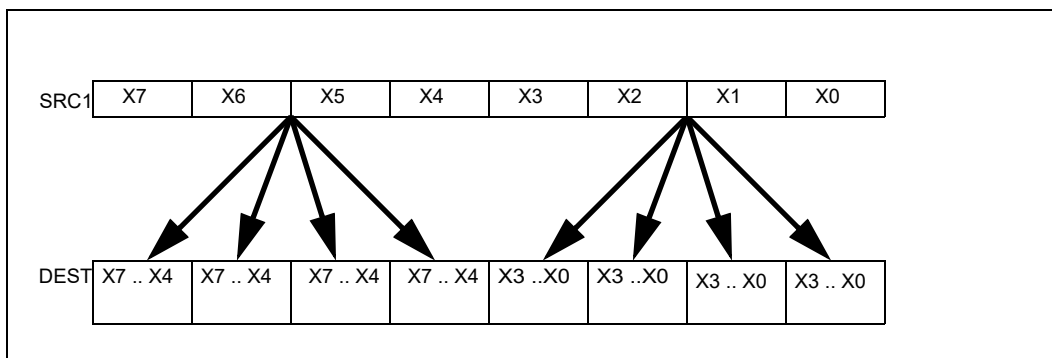
**Description**

(variable control version)

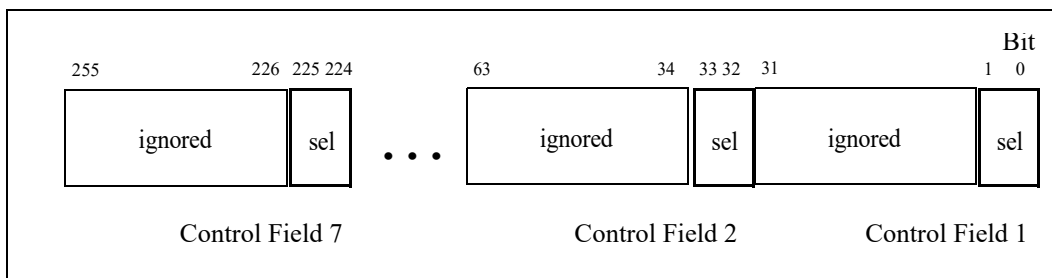
Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

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 &gt;= 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 Exceptions Type 4;

#UD If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD If either (E)VEX.vvvv != 1111B and with imm8.

## VPERMPD—Permute Double-Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 01 /r ib 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 VPERMPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	AVX512VL 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 VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	B	V/V	AVX512F	Permute double-precision floating-point elements in zmm2/m512/m64bcst using indexes in imm8 and store the result in zmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 16 /r VPERMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL 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 VPERMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Permute double-precision floating-point elements in zmm3/m512/m64bcst using indexes 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	NA	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

#### 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 Exceptions Type 4; additionally

```

#UD          If VEX.L = 0.
              If VEX.vvvv != 1111B.

```

EVEX-encoded instruction, see Exceptions Type E4NF.

```

#UD          If encoded with EVEX.128.
              If EVEX.vvvv != 1111B and with imm8.

```

## VPERMPS—Permute Single-Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 16 /r VPERMPS ymm1, ymm2, 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 VPERMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VPERMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute single-precision floating-point values in zmm3/m512/m32bcst using indexes in zmm2 and store the result in zmm1 subject to write mask k1.

### Instruction Operand Encoding

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

### 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 Exceptions Type 4; additionally

#UD                    If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E4NF.



## VPERMQ—Qwords Element Permutation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r ib 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 VPERMQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	AVX512VL 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 VPERMQ zmm1 {k1}{z}, 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 VPERMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1.
EVEX.512.66.0F38.W1 36 /r VPERMQ zmm1 {k1}{z}, zmm2, 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	NA	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
B	Full	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

#### 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] &gt;&gt; (IMM8[1:0] \* 64))[63:0];

TMP\_DEST[127:64] ← (TMP\_SRC[255:0] &gt;&gt; (IMM8[3:2] \* 64))[63:0];

TMP\_DEST[191:128] ← (TMP\_SRC[255:0] &gt;&gt; (IMM8[5:4] \* 64))[63:0];

TMP\_DEST[255:192] ← (TMP\_SRC[255:0] &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

**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] &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] &gt;&gt; (SRC1[66:64] \* 64))[63:0];

TMP\_DEST[191:128] ← (TMP\_SRC2[511:0] &gt;&gt; (SRC1[130:128] \* 64))[63:0];

TMP\_DEST[255:192] ← (TMP\_SRC2[511:0] &gt;&gt; (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 Exceptions Type 4; additionally

```

#UD          If VEX.L = 0.
             If VEX.vvvv != 1111B.

```

EVEX-encoded instruction, see Exceptions Type E4NF.

```

#UD          If encoded with EVEX.128.
             If EVEX.vvvv != 1111B and with imm8.

```

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 7D /r VPERMT2B xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL 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 VPERMT2B ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL 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 VPERMT2B zmm1 {k1}{z}, zmm2, 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)	NA

### Description

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

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

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

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

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

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

IF VL = 128:

id ← 3;

ELSE IF VL = 256:

id ← 4;

ELSE IF VL = 512:

id ← 5;

FI;

TMP\_DEST[VL-1:0] ← DEST[VL-1:0];

FOR j ← 0 TO KL-1

off ← 8\*SRC1[j\*8 + id; j\*8];

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

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

ELSE IF \*zeroing-masking\*

DEST[j\*8 + 7: j\*8] ← 0;

\*ELSE

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

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

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

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

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

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

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

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

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

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

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

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

**SIMD Floating-Point Exceptions**

None.

**Other Exceptions**

See Exceptions Type E4NF.nb.

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

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

## Instruction Operand Encoding

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

## Description

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[+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[+id:i]

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

THEN

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

THEN

DEST[+31:i] ← SRC1[+id+1] ? SRC2[31:0]

: TMP\_DEST[off+31:off]

ELSE

DEST[+31:i] ← SRC1[+id+1] ? SRC2[off+31:off]

: TMP\_DEST[off+31:off]

FI

ELSE

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

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

ELSE ; zeroing-masking

DEST[+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

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

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

IF VL = 128

id ← 0

FI;

IF VL = 256

id ← 1

FI;

IF VL = 512

id ← 2

FI;

TMP\_DEST ← DEST

FOR j ← 0 TO KL-1



```

i ← j * 64
off ← 64*SRC1[i+id:i]
IF k1[j] OR *no writemask*
  THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN
        DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[63:0]
        : TMP_DEST[off+63:off]
      ELSE
        DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[off+63:off]
        : TMP_DEST[off+63:off]
      FI
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+63:i] remains unchanged*
      ELSE ; zeroing-masking
        DEST[i+63:i] ← 0
      FI
    FI;
  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 Exceptions Type E4NF.

VPERMT2W: See Exceptions Type E4NF.nb.

## VPEXPANDD—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 89 /r VPEXPANDD xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 89 /r VPEXPANDD ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 89 /r VPEXPANDD zmm1 {k1}{z}, 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)	NA	NA

### 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)

$k \leftarrow 0$

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

$i \leftarrow j * 32$

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

THEN

DEST[ $i+31:i$ ]  $\leftarrow$  SRC[ $k+31:k$ ];

$k \leftarrow k + 32$

ELSE

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

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

ELSE ; zeroing-masking

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

FI

FI;

ENDFOR

DEST[ $MAXVL-1:VL$ ]  $\leftarrow$  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.

#UD If EVEX.vvvv != 1111B.

## VPEXPANDQ—Load Sparse Packed Quadword Integer Values from Dense Memory / Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 89 /r VPEXPANDQ xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 89 /r VPEXPANDQ ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 89 /r 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)	NA	NA

### 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)

$k \leftarrow 0$

FOR  $j \leftarrow 0$  TO KL-1

$i \leftarrow j * 64$

  IF  $k1[j]$  OR \*no writemask\*

    THEN

$DEST[i+63:i] \leftarrow SRC[k+63:k];$

$k \leftarrow k + 64$

    ELSE

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

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

      ELSE ; zeroing-masking

        THEN  $DEST[i+63:i] \leftarrow 0$

    FI

  FI;

ENDFOR

$DEST[MAXVL-1:VL] \leftarrow 0$

**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.

#UD If EVEX.vvvv != 1111B.

## VPGATHERDD/VPGATHERQD – Gather Packed Dword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 90 /r VPGATHERDD <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.128.66.0F38.W0 91 /r VPGATHERQD <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.256.66.0F38.W0 90 /r VPGATHERDD <i>ymm1</i> , <i>vm32y</i> , <i>ymm2</i>	RMV	V/V	AVX2	Using dword indices specified in <i>vm32y</i> , gather dword from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.256.66.0F38.W0 91 /r VPGATHERQD <i>xmm1</i> , <i>vm64y</i> , <i>xmm2</i>	RMV	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather dword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

### 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[i + 31:i] ← FFFFFFFFH; // extend from most significant bit

  ELSE

    MASK[i + 31:i] ← 0;

  FI;

ENDFOR

FOR j ← 0 to 3

  i ← j \* 32;

  DATA\_ADDR ← BASE\_ADDR + (SignExtend(VINDEX[i+31:i])\*SCALE + DISP;

  IF MASK[31:i] THEN

    DEST[i + 31:i] ← FETCH\_32BITS(DATA\_ADDR); // a fault exits the instruction

  FI;

  MASK[i + 31:i] ← 0;

ENDFOR

DEST[MAXVL-1:128] ← 0;



**VPGATHERQD (VEX.128 version)**

```

MASK[MAXVL-1:64] ← 0;
FOR j ← 0 to 3
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 1
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
DEST[MAXVL-1:64] ← 0;

```

**VPGATHERDD (VEX.256 version)**

```

MASK[MAXVL-1:256] ← 0;
FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31+i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 7
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[j+31:i])*SCALE + DISP;
  IF MASK[31+i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
DEST[MAXVL-1:256] ← 0;

```

**VPGATHERQD (VEX.256 version)**

```

MASK[MAXVL-1:128] ← 0;
FOR j ← 0 to 7
  i ← j * 32;
  IF MASK[31:i] THEN
    MASK[i + 31:i] ← FFFFFFFFH; // extend from most significant bit
  ELSE
    MASK[i + 31:i] ← 0;
  FI;
ENDFOR
FOR j ← 0 to 3
  k ← j * 64;
  i ← j * 32;
  DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
  IF MASK[31:i] THEN
    DEST[i + 31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
  FI;
  MASK[i + 31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;

```

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

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 Exceptions Type 12.

## VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword with Signed Dword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 90 /vsib VPGATHERDD xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 90 /vsib VPGATHERDD ymm1 {k1}, vm32y	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 90 /vsib 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 VPGATHERDQ xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 90 /vsib VPGATHERDQ ymm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 90 /vsib 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, VectorReg(R): VSIB:index	NA	NA

### 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], 1)

    k1[j] ← 0

    ELSE \*DEST[i+31:i] ← remains unchanged\*       ; Only merging masking is allowed

  FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAXVL-1:VL] ← 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 Exceptions Type E12.

## VPGATHERDQ/VPGATHERQQ – Gather Packed Qword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W1 90 /r VPGATHERDQ <i>xmm1</i> , <i>vm32x</i> , <i>xmm2</i>	A	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather qword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.128.66.0F38.W1 91 /r VPGATHERQQ <i>xmm1</i> , <i>vm64x</i> , <i>xmm2</i>	A	V/V	AVX2	Using qword indices specified in <i>vm64x</i> , gather qword values from memory conditioned on mask specified by <i>xmm2</i> . Conditionally gathered elements are merged into <i>xmm1</i> .
VEX.256.66.0F38.W1 90 /r VPGATHERDQ <i>ymm1</i> , <i>vm32x</i> , <i>ymm2</i>	A	V/V	AVX2	Using dword indices specified in <i>vm32x</i> , gather qword values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .
VEX.256.66.0F38.W1 91 /r VPGATHERQQ <i>ymm1</i> , <i>vm64y</i> , <i>ymm2</i>	A	V/V	AVX2	Using qword indices specified in <i>vm64y</i> , gather qword values from memory conditioned on mask specified by <i>ymm2</i> . Conditionally gathered elements are merged into <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

### Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

## Operation

DEST ← SRC1;

BASE\_ADDR: base register encoded in VSIB addressing;

VINDEX: the vector index register encoded by VSIB addressing;

SCALE: scale factor encoded by SIB:[7:6];

DISP: optional 1, 4 byte displacement;

MASK ← SRC3;

### VPGATHERDQ (VEX.128 version)

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;

**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[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

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 Exceptions Type 12.

## VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 91 /vsib VPGATHERQD 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 VPGATHERQQ xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 91 /vsib VPGATHERQQ ymm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 91 /vsib VPGATHERQQ 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, VectorReg(R): VSIB:index	NA	NA

### 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], 1)

        k1[j] ← 0

        ELSE \*DEST[i+31:i] ← remains unchanged\*           ; Only merging masking is allowed

    FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAXVL-1:VL/2] ← 0

#### VPGATHERQQ (EVEX encoded version)

(KL, VL) = (2, 64), (4, 128), (8, 256)

FOR j ← 0 TO KL-1

    i ← j \* 64

    IF k1[j]

        THEN DEST[i+63:i] ←

            MEM[BASE\_ADDR + (VINDEX[i+63:i]) \* SCALE + DISP]

        k1[j] ← 0

        ELSE \*DEST[i+63:i] ← remains unchanged\*           ; Only merging masking is allowed

    FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

DEST[MAXVL-1:VL] ← 0

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

```
VPGATHERQD __m256i __mm512_i64gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERQD __m256i __mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i __mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i __mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQQ __m512i __mm512_i64gather_epi64(__m512i vdx, void * base, int scale);
VPGATHERQQ __m512i __mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQQ __m256i __mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQQ __m128i __mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E12.

## VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 44 /r VPLZCNTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL 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 VPLZCNTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL 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 VPLZCNTD zmm1 {k1}{z}, 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 VPLZCNTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VPLZCNTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VPLZCNTQ zmm1 {k1}{z}, 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)	NA	NA

#### 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 Exceptions Type E4.

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

Opcode/ Instruction	Op/ En	32/64 bit Mode Support	CPUID	Description
EVEX.128.66.0F38.W1 B5 /r VPMADD52HUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512_IFMA 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 VPMADD52HUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B5 /r VPMADD52HUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the high 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

### Instruction Operand Encoding

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

### Description

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

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



## Operation

### VPMADD52HUQ (EVEX encoded)

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

FOR j ← 0 TO KL-1

  i ← j \* 64;

  IF k1[j] OR \*no writemask\* THEN

    IF src2 is Memory AND EVEX.b=1 THEN

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

    ELSE

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

    FI;

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

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

    DEST[i+63:i] ← Temp2[63:0];

  ELSE

    IF \*zeroing-masking\* THEN

      DEST[i+63:i] ← 0;

    ELSE \*merge-masking\*

      DEST[i+63:i] is unchanged;

    FI;

  FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0

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

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

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

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

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

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

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

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

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

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

### Flags Affected

None.

### SIMD Floating-Point Exceptions

None

### Other Exceptions

See Exceptions Type E4.

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

Opcode/ Instruction	Op/En	32/64 bit Mode Support	CPUID	Description
EVEX.128.66.0F38.W1 B4 /r VPMADD52LUQ xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	AVX512_IFMA 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 VPMADD52LUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512_IFMA AVX512VL	Multiply unsigned 52-bit integers in ymm2 and ymm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 B4 /r VPMADD52LUQ zmm1 {k1}{z}, zmm2,zmm3/m512/m64bcst	A	V/V	AVX512_IFMA	Multiply unsigned 52-bit integers in zmm2 and zmm3/m128 and add the low 52 bits of the 104-bit product to the qword unsigned integers in zmm1 using writemask k1.

### Instruction Operand Encoding

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

### Description

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

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

**Operation****VPMADD52LUQ (EVEX encoded)**

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

FOR j ← 0 TO KL-1

i ← j \* 64;

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

IF src2 is Memory AND EVEX.b=1 THEN

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

ELSE

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

FI;

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

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

DEST[i+63:i] ← Temp2[63:0];

ELSE

IF \*zeroing-masking\* THEN

DEST[i+63:i] ← 0;

ELSE \*merge-masking\*

DEST[i+63:i] is unchanged;

FI;

FI;

ENDFOR

DEST[MAX\_VL-1:VL] ← 0;

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

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

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

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

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

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

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

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

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

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

**Flags Affected**

None.

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E4.

## VPMASKMOV – Conditional SIMD Integer Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 8C /r VPMASKMOVD <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX2	Conditionally load dword values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.256.66.0F38.W0 8C /r VPMASKMOVD <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX2	Conditionally load dword values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.128.66.0F38.W1 8C /r VPMASKMOVQ <i>xmm1, xmm2, m128</i>	RVM	V/V	AVX2	Conditionally load qword values from <i>m128</i> using mask in <i>xmm2</i> and store in <i>xmm1</i> .
VEX.256.66.0F38.W1 8C /r VPMASKMOVQ <i>ymm1, ymm2, m256</i>	RVM	V/V	AVX2	Conditionally load qword values from <i>m256</i> using mask in <i>ymm2</i> and store in <i>ymm1</i> .
VEX.128.66.0F38.W0 8E /r VPMASKMOVD <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX2	Conditionally store dword values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.256.66.0F38.W0 8E /r VPMASKMOVD <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX2	Conditionally store dword values from <i>ymm2</i> using mask in <i>ymm1</i> .
VEX.128.66.0F38.W1 8E /r VPMASKMOVQ <i>m128, xmm1, xmm2</i>	MVR	V/V	AVX2	Conditionally store qword values from <i>xmm2</i> using mask in <i>xmm1</i> .
VEX.256.66.0F38.W1 8E /r VPMASKMOVQ <i>m256, ymm1, ymm2</i>	MVR	V/V	AVX2	Conditionally store qword values from <i>ymm2</i> using mask in <i>ymm1</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv	ModRM:reg (r)	NA

### 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 Exceptions Type 6 (No AC# reported for any mask bit combinations).

## VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 29 /r VPMOVB2M k1, xmm1	RM	V/V	AVX512VL 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 VPMOVB2M k1, ymm1	RM	V/V	AVX512VL 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 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 VPMOVW2M k1, xmm1	RM	V/V	AVX512VL 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 VPMOVW2M k1, ymm1	RM	V/V	AVX512VL 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 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 VPMOVD2M k1, xmm1	RM	V/V	AVX512VL 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 VPMOVD2M k1, ymm1	RM	V/V	AVX512VL 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 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 VPMOVQ2M k1, xmm1	RM	V/V	AVX512VL 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 VPMOVQ2M k1, ymm1	RM	V/V	AVX512VL 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 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)	NA	NA

### Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

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

**Operation****VPMOVB2M (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 8

IF SRC[i+7]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPMOVW2M (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 16

IF SRC[i+15]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPMOVD2M (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

IF SRC[i+31]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPMOVQ2M (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

IF SRC[i+63]

THEN DEST[j] ← 1

ELSE DEST[j] ← 0

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0



**Intel C/C++ Compiler Intrinsic Equivalents**

```

VPMPOVB2M __mmask64 _mm512_movepi8_mask( __m512i );
VPMPOVD2M __mmask16 _mm512_movepi32_mask( __m512i );
VPMPOVQ2M __mmask8 _mm512_movepi64_mask( __m512i );
VPMPOVW2M __mmask32 _mm512_movepi16_mask( __m512i );
VPMPOVB2M __mmask32 _mm256_movepi8_mask( __m256i );
VPMPOVD2M __mmask8 _mm256_movepi32_mask( __m256i );
VPMPOVQ2M __mmask8 _mm256_movepi64_mask( __m256i );
VPMPOVW2M __mmask16 _mm256_movepi16_mask( __m256i );
VPMPOVB2M __mmask16 _mm_movepi8_mask( __m128i );
VPMPOVD2M __mmask8 _mm_movepi32_mask( __m128i );
VPMPOVQ2M __mmask8 _mm_movepi64_mask( __m128i );
VPMPOVW2M __mmask8 _mm_movepi16_mask( __m128i );

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E7NM

#UD                    If EVEX.vvvv != 1111B.

## VPMOVD/VPMSVDB/VPMOVUSDB—Down Convert DWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 31 /r VPMOVD <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed double-word integers from <i>xmm2</i> into 4 packed byte integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 21 /r VPMSVDB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed double-word integers from <i>xmm2</i> into 4 packed signed byte integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 11 /r VPMOVUSDB <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned double-word integers from <i>xmm2</i> into 4 packed unsigned byte integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 31 /r VPMOVD <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed double-word integers from <i>ymm2</i> into 8 packed byte integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 21 /r VPMSVDB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed signed double-word integers from <i>ymm2</i> into 8 packed signed byte integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 11 /r VPMOVUSDB <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed unsigned double-word integers from <i>ymm2</i> into 8 packed unsigned byte integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 31 /r VPMOVD <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	A	V/V	AVX512F	Converts 16 packed double-word integers from <i>zmm2</i> into 16 packed byte integers in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 21 /r VPMSVDB <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	A	V/V	AVX512F	Converts 16 packed signed double-word integers from <i>zmm2</i> into 16 packed signed byte integers in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	A	V/V	AVX512F	Converts 16 packed unsigned double-word integers from <i>zmm2</i> into 16 packed unsigned byte integers in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> .

### 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)	NA	NA

### Description

VPMOVD down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMSVDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVUSDB 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****VPMOVD B 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;

```

**VPMOVD B 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

```

VPMOVB __m128i __mm512_cvtepi32_epi8(__m512i a);
VPMOVB __m128i __mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVB __m128i __mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVB 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);
VPMOVB __m128i __mm256_cvtepi32_epi8(__m256i a);
VPMOVB __m128i __mm256_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVB __m128i __mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i b);
VPMOVB void __mm256_mask_cvtepi32_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVB __m128i __mm_cvtepi32_epi8(__m128i a);
VPMOVB __m128i __mm_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVB __m128i __mm_maskz_cvtepi32_epi8(__mmask8 k, __m128i b);
VPMOVB void __mm_mask_cvtepi32_storeu_epi8(void *, __mmask8 k, __m128i b);

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert Dword to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 33 /r VPMOVDW <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed double-word integers from <i>xmm2</i> into 4 packed word integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 23 /r VPMOVSDW <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed double-word integers from <i>xmm2</i> into 4 packed signed word integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 13 /r VPMOVUSDW <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned double-word integers from <i>xmm2</i> into 4 packed unsigned word integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 33 /r VPMOVDW <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed double-word integers from <i>ymm2</i> into 8 packed word integers in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 23 /r VPMOVSDW <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed signed double-word integers from <i>ymm2</i> into 8 packed signed word integers in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 13 /r VPMOVUSDW <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed unsigned double-word integers from <i>ymm2</i> into 8 packed unsigned word integers in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 33 /r VPMOVDW <i>ymm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 16 packed double-word integers from <i>zmm2</i> into 16 packed word integers in <i>ymm1/m256</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 23 /r VPMOVSDW <i>ymm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 16 packed signed double-word integers from <i>zmm2</i> into 16 packed signed word integers in <i>ymm1/m256</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW <i>ymm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 16 packed unsigned double-word integers from <i>zmm2</i> into 16 packed unsigned word integers in <i>ymm1/m256</i> using unsigned saturation under writemask <i>k1</i> .

### 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)	NA	NA

### Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

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

**Operation****VPMOVDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

```

**VPMOVDW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
    ELSE
      *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVSDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
    ELSE
      IF *merging-masking* ; merging-masking
        THEN *DEST[i+15:i] remains unchanged*
        ELSE *zeroing-masking* ; zeroing-masking
          DEST[i+15:i] ← 0
      FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

```

**VPMOVSdW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```

**VPMOVUSDW instruction (EVEX encoded versions) when dest is a register**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+15:i] remains unchanged*
    ELSE *zeroing-masking* ; zeroing-masking
      DEST[i+15:i] ← 0
    FI
  FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;

```

**VPMOVUSDW instruction (EVEX encoded versions) when dest is memory**

```

(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
  i ← j * 16
  m ← j * 32
  IF k1[j] OR *no writemask*
    THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
  ELSE
    *DEST[i+15:i] remains unchanged* ; merging-masking
  FI;
ENDFOR

```



### Intel C/C++ Compiler Intrinsic Equivalents

```

VPMOVDW __m256i _mm512_cvtepi32_epi16(__m512i a);
VPMOVDW __m256i _mm512_mask_cvtepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_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 Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EEX.512.F3.0F38.W0 28 /r 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.
EEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EEX.512.F3.0F38.W1 28 /r 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.
EEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EEX.512.F3.0F38.W0 38 /r 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.
EEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EEX.512.F3.0F38.W1 38 /r 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)	NA	NA

### 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.

EEX.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 Exceptions Type E7NM

#UD                    If EVEX.vvvv != 1111B.

## VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 32 /r VPMOVQB <i>xmm1/m16 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed byte integers in <i>xmm1/m16</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 22 /r VPMOVSQB <i>xmm1/m16 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed signed quad-word integers from <i>xmm2</i> into 2 packed signed byte integers in <i>xmm1/m16</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB <i>xmm1/m16 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned byte integers in <i>xmm1/m16</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 32 /r VPMOVQB <i>xmm1/m32 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed byte integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 22 /r VPMOVSQB <i>xmm1/m32 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed byte integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB <i>xmm1/m32 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned byte integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 32 /r VPMOVQB <i>xmm1/m64 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed byte integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 22 /r VPMOVSQB <i>xmm1/m64 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed byte integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB <i>xmm1/m64 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned byte integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .

### 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)	NA	NA

### 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****VPMOVB 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;

```

**VPMOVB 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

```

**VPMOVB 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**

```

VPMOVQB __m128i __mm512_cvtepi64_epi8( __m512i a);
VPMOVQB __m128i __mm512_mask_cvtepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i __mm512_maskz_cvtepi64_epi8( __mmask8 k, __m512i a);
VPMOVQB void __mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVSQB __m128i __mm512_cvtsepi64_epi8( __m512i a);
VPMOVSQB __m128i __mm512_mask_cvtsepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVSQB __m128i __mm512_maskz_cvtsepi64_epi8( __mmask8 k, __m512i a);
VPMOVSQB void __mm512_mask_cvtsepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i __mm512_cvtusepi64_epi8( __m512i a);
VPMOVUSQB __m128i __mm512_mask_cvtusepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQB __m128i __mm512_maskz_cvtusepi64_epi8( __mmask8 k, __m512i a);
VPMOVUSQB void __mm512_mask_cvtusepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i __mm256_cvtusepi64_epi8(__m256i a);
VPMOVUSQB __m128i __mm256_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQB __m128i __mm256_maskz_cvtusepi64_epi8( __mmask8 k, __m256i b);
VPMOVUSQB void __mm256_mask_cvtusepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVUSQB __m128i __mm_cvtusepi64_epi8(__m128i a);
VPMOVUSQB __m128i __mm_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQB __m128i __mm_maskz_cvtusepi64_epi8( __mmask8 k, __m128i b);
VPMOVUSQB void __mm_mask_cvtusepi64_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVSQB __m128i __mm256_cvtsepi64_epi8(__m256i a);
VPMOVSQB __m128i __mm256_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVSQB __m128i __mm256_maskz_cvtsepi64_epi8( __mmask8 k, __m256i b);
VPMOVSQB void __mm256_mask_cvtsepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVSQB __m128i __mm_cvtsepi64_epi8(__m128i a);
VPMOVSQB __m128i __mm_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSQB __m128i __mm_maskz_cvtsepi64_epi8( __mmask8 k, __m128i b);
VPMOVSQB void __mm_mask_cvtsepi64_storeu_epi8(void *, __mmask8 k, __m128i b);
VPMOVQB __m128i __mm256_cvtepi64_epi8(__m256i a);
VPMOVQB __m128i __mm256_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVQB __m128i __mm256_maskz_cvtepi64_epi8( __mmask8 k, __m256i b);
VPMOVQB void __mm256_mask_cvtepi64_storeu_epi8(void *, __mmask8 k, __m256i b);
VPMOVQB __m128i __mm_cvtepi64_epi8(__m128i a);
VPMOVQB __m128i __mm_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVQB __m128i __mm_maskz_cvtepi64_epi8( __mmask8 k, __m128i b);
VPMOVQB void __mm_mask_cvtepi64_storeu_epi8(void *, __mmask8 k, __m128i b);

```

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

EVEX-encoded instruction, see Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.



## VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

Opcode/ Instruction	Op / En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 35 /r VPMOVQD <i>xmm1/m128 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed double-word integers in <i>xmm1/m128</i> with truncation subject to writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 25 /r VPMOVSQD <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed signed quad-word integers from <i>xmm2</i> into 2 packed signed double-word integers in <i>xmm1/m64</i> using signed saturation subject to writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 15 /r VPMOVUSQD <i>xmm1/m64 {k1}{z}, xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned double-word integers in <i>xmm1/m64</i> using unsigned saturation subject to writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 35 /r VPMOVQD <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed double-word integers in <i>xmm1/m128</i> with truncation subject to writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 25 /r VPMOVSQD <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed double-word integers in <i>xmm1/m128</i> using signed saturation subject to writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 15 /r VPMOVUSQD <i>xmm1/m128 {k1}{z}, ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned double-word integers in <i>xmm1/m128</i> using unsigned saturation subject to writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 35 /r VPMOVQD <i>ymm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed double-word integers in <i>ymm1/m256</i> with truncation subject to writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 25 /r VPMOVSQD <i>ymm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed double-word integers in <i>ymm1/m256</i> using signed saturation subject to writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 15 /r VPMOVUSQD <i>ymm1/m256 {k1}{z}, zmm2</i>	A	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned double-word integers in <i>ymm1/m256</i> using unsigned saturation subject to writemask <i>k1</i> .

### 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)	NA	NA

### 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****VPMOVD 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;

```

**VPMOVD 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);
VPMOVUSD __m256i __mm512_cvtusepi64_epi32( __m512i a);
VPMOVUSD __m256i __mm512_mask_cvtusepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVUSD __m256i __mm512_maskz_cvtusepi64_epi32( __mmask8 k, __m512i a);
VPMOVUSD void __mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSD __m128i __mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSD __m128i __mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSD __m128i __mm256_maskz_cvtusepi64_epi32( __mmask8 k, __m256i b);
VPMOVUSD void __mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSD __m128i __mm_cvtusepi64_epi32(__m128i a);
VPMOVUSD __m128i __mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSD __m128i __mm_maskz_cvtusepi64_epi32( __mmask8 k, __m128i b);
VPMOVUSD 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 Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.

## VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 34 /r VPMOVQW <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from <i>xmm2</i> into 2 packed word integers in <i>xmm1/m32</i> with truncation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 24 /r VPMOVSQW <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed word integers in <i>xmm1/m32</i> using signed saturation under writemask <i>k1</i> .
EVEX.128.F3.0F38.W0 14 /r VPMOVUSQW <i>xmm1/m32</i> { <i>k1</i> }{ <i>z</i> }, <i>xmm2</i>	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from <i>xmm2</i> into 2 packed unsigned word integers in <i>xmm1/m32</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 34 /r VPMOVQW <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from <i>ymm2</i> into 4 packed word integers in <i>xmm1/m64</i> with truncation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 24 /r VPMOVSQW <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from <i>ymm2</i> into 4 packed signed word integers in <i>xmm1/m64</i> using signed saturation under writemask <i>k1</i> .
EVEX.256.F3.0F38.W0 14 /r VPMOVUSQW <i>xmm1/m64</i> { <i>k1</i> }{ <i>z</i> }, <i>ymm2</i>	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from <i>ymm2</i> into 4 packed unsigned word integers in <i>xmm1/m64</i> using unsigned saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 34 /r VPMOVQW <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	A	V/V	AVX512F	Converts 8 packed quad-word integers from <i>zmm2</i> into 8 packed word integers in <i>xmm1/m128</i> with truncation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 24 /r VPMOVSQW <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	A	V/V	AVX512F	Converts 8 packed signed quad-word integers from <i>zmm2</i> into 8 packed signed word integers in <i>xmm1/m128</i> using signed saturation under writemask <i>k1</i> .
EVEX.512.F3.0F38.W0 14 /r VPMOVUSQW <i>xmm1/m128</i> { <i>k1</i> }{ <i>z</i> }, <i>zmm2</i>	A	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from <i>zmm2</i> into 8 packed unsigned word integers in <i>xmm1/m128</i> using unsigned saturation under writemask <i>k1</i> .

### 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)	NA	NA

### 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 Exceptions Type E6.

#UD If EVEX.vvvv != 1111B.



## VPMOVB/WB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 30 /r VPMOVBWB <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>	A	V/V	AVX512VL AVX512BW	Converts 8 packed word integers from <i>xmm2</i> into 8 packed bytes in <i>xmm1/m64</i> with truncation under writemask k1.
EVEX.128.F3.0F38.W0 20 /r VPMOVSWB <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>	A	V/V	AVX512VL AVX512BW	Converts 8 packed signed word integers from <i>xmm2</i> into 8 packed signed bytes in <i>xmm1/m64</i> using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 10 /r VPMOVUSWB <i>xmm1/m64</i> {k1}{z}, <i>xmm2</i>	A	V/V	AVX512VL AVX512BW	Converts 8 packed unsigned word integers from <i>xmm2</i> into 8 packed unsigned bytes in <i>xmm1/m64</i> using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 30 /r VPMOVBWB <i>xmm1/m128</i> {k1}{z}, <i>ymm2</i>	A	V/V	AVX512VL AVX512BW	Converts 16 packed word integers from <i>ymm2</i> into 16 packed bytes in <i>xmm1/m128</i> with truncation under writemask k1.
EVEX.256.F3.0F38.W0 20 /r VPMOVSWB <i>xmm1/m128</i> {k1}{z}, <i>ymm2</i>	A	V/V	AVX512VL AVX512BW	Converts 16 packed signed word integers from <i>ymm2</i> into 16 packed signed bytes in <i>xmm1/m128</i> using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 10 /r VPMOVUSWB <i>xmm1/m128</i> {k1}{z}, <i>ymm2</i>	A	V/V	AVX512VL AVX512BW	Converts 16 packed unsigned word integers from <i>ymm2</i> into 16 packed unsigned bytes in <i>xmm1/m128</i> using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 30 /r VPMOVBWB <i>ymm1/m256</i> {k1}{z}, <i>zmm2</i>	A	V/V	AVX512BW	Converts 32 packed word integers from <i>zmm2</i> into 32 packed bytes in <i>ymm1/m256</i> with truncation under writemask k1.
EVEX.512.F3.0F38.W0 20 /r VPMOVSWB <i>ymm1/m256</i> {k1}{z}, <i>zmm2</i>	A	V/V	AVX512BW	Converts 32 packed signed word integers from <i>zmm2</i> into 32 packed signed bytes in <i>ymm1/m256</i> using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 10 /r VPMOVUSWB <i>ymm1/m256</i> {k1}{z}, <i>zmm2</i>	A	V/V	AVX512BW	Converts 32 packed unsigned word integers from <i>zmm2</i> into 32 packed unsigned bytes in <i>ymm1/m256</i> 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)	NA	NA

#### 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 Exceptions Type E6.

#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 VPMULTISHIFTQB xmm1 {k1}{z}, xmm2,xmm3/m128/m64bcst	A	V/V	AVX512_VBMI 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 VPMULTISHIFTQB ymm1 {k1}{z}, ymm2,ymm3/m256/m64bcst	A	V/V	AVX512_VBMI 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 VPMULTISHIFTQB zmm1 {k1}{z}, 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)	NA

#### Description

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

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

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

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

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

FOR i ← 0 TO KL-1

IF EVEX.b=1 AND src2 is memory THEN

tcur ← src2.qword[0]; //broadcasting

ELSE

tcur ← src2.qword[i];

FI;

FOR j ← 0 to 7

ctrl ← src1.qword[i].byte[j] &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 Exceptions Type E4NF.

## VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 15 /r VPROLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VPROLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.128.66.0F38.W1 15 /r VPROLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VPROLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 15 /r VPROLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VPROLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.256.66.0F38.W1 15 /r VPROLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VPROLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 15 /r VPROLVD zmm1 {k1}{z}, zmm2, 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 VPROLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512F	Rotate left of doublewords in zmm3/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.
EVEX.512.66.0F38.W1 15 /r VPROLVQ zmm1 {k1}{z}, zmm2, 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 VPROLQ zmm1 {k1}{z}, 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	NA
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

**Operation**

```
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC << COUNT) | (SRC >> (32 - COUNT));
```

```
LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC << COUNT) | (SRC >> (64 - COUNT));
```

**VPROLD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\* THEN

    IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

      THEN DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[31:0], imm8)

      ELSE DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[i+31:i], imm8)

    FI;

  ELSE

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

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

    ELSE \*zeroing-masking\* ; zeroing-masking

      DEST[i+31:i] ← 0

    FI

  FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0



**VPROLVD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

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

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

THEN DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[31:0])

ELSE DEST[i+31:i] ← LEFT\_ROTATE\_DWORDS(SRC1[i+31:i], SRC2[i+31:i])

FI;

ELSE

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

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

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPROLQ (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

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

IF (EVEX.b = 1) AND (SRC1 \*is memory\*)

THEN DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[63:0], imm8)

ELSE DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], imm8)

FI;

ELSE

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

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

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] ← 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

**VPROLVQ (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

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

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

THEN DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[63:0])

ELSE DEST[i+63:i] ← LEFT\_ROTATE\_QWORDS(SRC1[i+63:i], SRC2[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 Exceptions Type E4.

## VPRORD/VPRORVD/VPRORQ/VPRORVQ—Bit Rotate Right

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 14 /r VPRORVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VPRORD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.
EVEX.128.66.0F38.W1 14 /r VPRORVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VPRORQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.
EVEX.256.66.0F38.W0 14 /r VPRORVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VPRORD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.
EVEX.256.66.0F38.W1 14 /r VPRORVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VPRORQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.
EVEX.512.66.0F38.W0 14 /r VPRORVD zmm1 {k1}{z}, zmm2, 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 VPRORD zmm1 {k1}{z}, 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 VPRORVQ zmm1 {k1}{z}, zmm2, 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 VPRORQ zmm1 {k1}{z}, 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	NA
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## 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;

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

**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[j+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[j+63:i], SRC2[63:0])

ELSE DEST[j+63:i] ← RIGHT\_ROTATE\_QWORDS(SRC1[j+63:i], SRC2[j+63:i])

FI;

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: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 Exceptions Type E4.



## VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A0 /vsib VPSCATTERDD vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A0 /vsib VPSCATTERDD vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A0 /vsib VPSCATTERDD vm32z {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32z {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.128.66.0F38.W0 A1 /vsib VPSCATTERQD vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A1 /vsib VPSCATTERQD vm64y {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A1 /vsib 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 VPSCATTERQQ vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A1 /vsib 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, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

#### Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address `BASE_ADDR` and index vector `VINDEX`, with scale `SCALE`. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.

- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  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 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  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

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

THEN  $\text{MEM}[\text{BASE\_ADDR} + \text{SignExtend}(\text{VINDEX}[i+31:i]) * \text{SCALE} + \text{DISP}] \leftarrow \text{SRC}[i+31:i]$

$k1[j] \leftarrow 0$

FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$

#### VPSCATTERDQ (EVEX encoded versions)

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

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

$i \leftarrow j * 64$

$k \leftarrow j * 32$

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

THEN  $\text{MEM}[\text{BASE\_ADDR} + \text{SignExtend}(\text{VINDEX}[k+31:k]) * \text{SCALE} + \text{DISP}] \leftarrow \text{SRC}[i+63:i]$

$k1[j] \leftarrow 0$

FI;

ENDFOR

$k1[\text{MAX\_KL}-1:KL] \leftarrow 0$

**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[i+31:i]

k1[j] ← 0

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

**VPSCATTERQQ (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN MEM[BASE\_ADDR + (VINDEK[j+63:j]) \* SCALE + DISP] ← SRC[i+63:i]

FI;

ENDFOR

k1[MAX\_KL-1:KL] ← 0

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

VPSCATTERDD void \_\_mm512\_i32scatter\_epi32(void \* base, \_\_m512i vdx, \_\_m512i a, int scale);

VPSCATTERDD void \_\_mm256\_i32scatter\_epi32(void \* base, \_\_m256i vdx, \_\_m256i a, int scale);

VPSCATTERDD void \_\_mm\_i32scatter\_epi32(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERDD void \_\_mm512\_mask\_i32scatter\_epi32(void \* base, \_\_mmask16 k, \_\_m512i vdx, \_\_m512i a, int scale);

VPSCATTERDD void \_\_mm256\_mask\_i32scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m256i a, int scale);

VPSCATTERDD void \_\_mm\_mask\_i32scatter\_epi32(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERDQ void \_\_mm512\_i32scatter\_epi64(void \* base, \_\_m256i vdx, \_\_m512i a, int scale);

VPSCATTERDQ void \_\_mm256\_i32scatter\_epi64(void \* base, \_\_m128i vdx, \_\_m256i a, int scale);

VPSCATTERDQ void \_\_mm\_i32scatter\_epi64(void \* base, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERDQ void \_\_mm512\_mask\_i32scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m256i vdx, \_\_m512i a, int scale);

VPSCATTERDQ void \_\_mm256\_mask\_i32scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m256i a, int scale);

VPSCATTERDQ void \_\_mm\_mask\_i32scatter\_epi64(void \* base, \_\_mmask8 k, \_\_m128i vdx, \_\_m128i a, int scale);

VPSCATTERQD void \_\_mm512\_i64scatter\_epi32(void \* base, \_\_m512i vdx, \_\_m256i a, int scale);

VPSCATTERQD void \_\_mm256\_i64scatter\_epi32(void \* base, \_\_m256i vdx, \_\_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 Exceptions Type E12.

## VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 47 /r 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 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 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 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 VPSLLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	AVX512VL 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 VPSLLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	AVX512VL 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 VPSLLVW zmm1 {k1}{z}, zmm2, 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 VPSLLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL 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 VPSLLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL 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 VPSLLVD zmm1 {k1}{z}, zmm2, 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 VPSLLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL 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 VPSLLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL 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 VPSLLVQ zmm1 {k1}{z}, zmm2, 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	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**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 Exceptions Type 4.  
 EVEX-encoded VPSLLVD/VPSLLVQ, see Exceptions Type E4.  
 EVEX-encoded VPSLLVW, see Exceptions Type E4.nb.

## VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 46 /r 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 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 VPSRAVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	AVX512VL 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 VPSRAVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	AVX512VL 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 VPSRAVW zmm1 {k1}{z}, zmm2, 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 VPSRAVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL 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 VPSRAVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL 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 VPSRAVD zmm1 {k1}{z}, zmm2, 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 VPSRAVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL 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 VPSRAVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL 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 VPSRAVQ zmm1 {k1}{z}, zmm2, 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	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element is filled with the corresponding sign bit of the source element.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

## Operation

### VPSRAVW (EVEX encoded version)

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

```

FOR j ← 0 TO KL-1
  i ← j * 16
  IF k1[j] OR *no writemask*
    THEN
      COUNT ← SRC2[i+3:i]
      IF COUNT < 16
        THEN DEST[i+15:i] ← SignExtend(SRC1[i+15:i] >> COUNT)
        ELSE
          FOR k ← 0 TO 15
            DEST[i+k] ← SRC1[i+15]
          ENDFOR;
        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[i+4:i]
          IF COUNT < 32
            THEN DEST[i+31:i] ← SignExtend(SRC1[i+31:i] >> COUNT)
            ELSE
              FOR k ← 0 TO 31
                DEST[i+k] ← SRC1[i+31]
              ENDFOR;
            FI
          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[i+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 Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

## VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 45 /r 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 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 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 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 VPSRLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	AVX512VL 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 VPSRLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	AVX512VL 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 VPSRLVW zmm1 {k1}{z}, zmm2, 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 VPSRLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL 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 VPSRLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL 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 VPSRLVD zmm1 {k1}{z}, zmm2, 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 VPSRLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL 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 VPSRLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL 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 VPSRLVQ zmm1 {k1}{z}, zmm2, 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	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**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 Exceptions Type 4.  
 EVEX-encoded VPSRLVD/Q, see Exceptions Type E4.  
 EVEX-encoded VPSRLVW, see Exceptions Type E4.nb.

## VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 25 /r ib VPTERNLOGD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL 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 VPTERNLOGD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL 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 VPTERNLOGD zmm1 {k1}{z}, zmm2, 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 VPTERNLOGQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL 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 VPTERNLOGQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL 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 VPTERNLOGQ zmm1 {k1}{z}, zmm2, 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 5-9 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

**Table 5-9. 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 Exceptions Type E4.

## VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 26 /r VPTESTMB k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL 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 VPTESTMB k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL 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 VPTESTMB k2 {k1}, zmm2, 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 VPTESTMW k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL 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 VPTESTMW k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL 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 VPTESTMW k2 {k1}, zmm2, 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 VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL 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 VPTESTMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL 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 VPTESTMD k2 {k1}, zmm2, 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 VPTESTMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL 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 VPTESTMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL 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 VPTESTMQ k2 {k1}, zmm2, 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)	NA
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

**Description**

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

**VPTESTMD/VPTESTMQ:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

**VPTESTMB/VPTESTMW:** The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

**Operation****VPTESTMB (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

  i ← j \* 8

  IF k1[j] OR \*no writemask\*

    THEN DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;

    ELSE DEST[j] = 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTMW (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

  i ← j \* 16

  IF k1[j] OR \*no writemask\*

    THEN DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;

    ELSE DEST[j] = 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTMD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

  i ← j \* 32

  IF k1[j] OR \*no writemask\*

    THEN

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

        THEN DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;

        ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;

      FI;

    ELSE DEST[j] ← 0 ; zeroing-masking only

  FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0



**VPTESTMQ (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

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

THEN

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

THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;

ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;

FI;

ELSE DEST[j] ← 0 ; zeroing-masking only

FI;

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**Intel C/C++ Compiler Intrinsic Equivalents**

VPTESTMB \_\_mmask64\_mm512\_test\_epi8\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMB \_\_mmask64\_mm512\_mask\_test\_epi8\_mask(\_\_mmask64, \_\_m512i a, \_\_m512i b);

VPTESTMW \_\_mmask32\_mm512\_test\_epi16\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMW \_\_mmask32\_mm512\_mask\_test\_epi16\_mask(\_\_mmask32, \_\_m512i a, \_\_m512i b);

VPTESTMD \_\_mmask16\_mm512\_test\_epi32\_mask( \_\_m512i a, \_\_m512i b);

VPTESTMD \_\_mmask16\_mm512\_mask\_test\_epi32\_mask(\_\_mmask16, \_\_m512i a, \_\_m512i b);

VPTESTMQ \_\_mmask8\_mm512\_test\_epi64\_mask(\_\_m512i a, \_\_m512i b);

VPTESTMQ \_\_mmask8\_mm512\_mask\_test\_epi64\_mask(\_\_mmask8, \_\_m512i a, \_\_m512i b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

VPTESTMD/Q: See Exceptions Type E4.

VPTESTMB/W: See Exceptions Type E4.nb.

## VPTESTNMB/W/D/Q—Logical NAND and Set

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID	Description
EVEX.128.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512F AVX512BW	Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512F AVX512BW	Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.128.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.256.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.512.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

## 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)	NA
B	Full	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

## Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

## Operation

## VPTESTNMB

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

FOR j ← 0 TO KL-1

    i ← j\*8

    IF MaskBit(j) OR \*no writemask\*

        THEN

            DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0

        ELSE DEST[j] ← 0; zeroing masking only

    FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

## VPTESTNMW

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

FOR j ← 0 TO KL-1

    i ← j\*16

    IF MaskBit(j) OR \*no writemask\*

        THEN

            DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0

        ELSE DEST[j] ← 0; zeroing masking only

    FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTNMD**

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

FOR j ← 0 TO KL-1

i ← j\*32

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

THEN

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

THEN DEST[i+31:i] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0

ELSE DEST[i] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0

FI

ELSE DEST[i] ← 0; zeroing masking only

FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

**VPTESTNMQ**

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

FOR j ← 0 TO KL-1

i ← j\*64

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

THEN

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

THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;

ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;

FI;

ELSE DEST[j] ← 0; zeroing masking only

FI

ENDFOR

DEST[MAX\_KL-1:KL] ← 0

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

VPTESTNMB \_\_mmask64 \_\_mm512\_testn\_epi8\_mask(\_\_m512i a, \_\_m512i b);

VPTESTNMB \_\_mmask64 \_\_mm512\_mask\_testn\_epi8\_mask(\_\_mmask64, \_\_m512i a, \_\_m512i b);

VPTESTNMB \_\_mmask32 \_\_mm256\_testn\_epi8\_mask(\_\_m256i a, \_\_m256i b);

VPTESTNMB \_\_mmask32 \_\_mm256\_mask\_testn\_epi8\_mask(\_\_mmask32, \_\_m256i a, \_\_m256i b);

VPTESTNMB \_\_mmask16 \_\_mm\_testn\_epi8\_mask(\_\_m128i a, \_\_m128i b);

VPTESTNMB \_\_mmask16 \_\_mm\_mask\_testn\_epi8\_mask(\_\_mmask16, \_\_m128i a, \_\_m128i b);

VPTESTNMW \_\_mmask32 \_\_mm512\_testn\_epi16\_mask(\_\_m512i a, \_\_m512i b);

VPTESTNMW \_\_mmask32 \_\_mm512\_mask\_testn\_epi16\_mask(\_\_mmask32, \_\_m512i a, \_\_m512i b);

VPTESTNMW \_\_mmask16 \_\_mm256\_testn\_epi16\_mask(\_\_m256i a, \_\_m256i b);

VPTESTNMW \_\_mmask16 \_\_mm256\_mask\_testn\_epi16\_mask(\_\_mmask16, \_\_m256i a, \_\_m256i b);

VPTESTNMW \_\_mmask8 \_\_mm\_testn\_epi16\_mask(\_\_m128i a, \_\_m128i b);

VPTESTNMW \_\_mmask8 \_\_mm\_mask\_testn\_epi16\_mask(\_\_mmask8, \_\_m128i a, \_\_m128i b);

VPTESTNMD \_\_mmask16 \_\_mm512\_testn\_epi32\_mask(\_\_m512i a, \_\_m512i b);

VPTESTNMD \_\_mmask16 \_\_mm512\_mask\_testn\_epi32\_mask(\_\_mmask16, \_\_m512i a, \_\_m512i b);

VPTESTNMD \_\_mmask8 \_\_mm256\_testn\_epi32\_mask(\_\_m256i a, \_\_m256i b);

VPTESTNMD \_\_mmask8 \_\_mm256\_mask\_testn\_epi32\_mask(\_\_mmask8, \_\_m256i a, \_\_m256i b);

VPTESTNMD \_\_mmask8 \_\_mm\_testn\_epi32\_mask(\_\_m128i a, \_\_m128i b);

VPTESTNMD \_\_mmask8 \_\_mm\_mask\_testn\_epi32\_mask(\_\_mmask8, \_\_m128i a, \_\_m128i b);

VPTESTNMQ \_\_mmask8 \_\_mm512\_testn\_epi64\_mask(\_\_m512i a, \_\_m512i b);

VPTESTNMQ \_\_mmask8 \_\_mm512\_mask\_testn\_epi64\_mask(\_\_mmask8, \_\_m512i a, \_\_m512i b);

VPTESTNMQ \_\_mmask8 \_\_mm256\_testn\_epi64\_mask(\_\_m256i a, \_\_m256i b);

VPTESTNMQ \_\_mmask8 \_\_mm256\_mask\_testn\_epi64\_mask(\_\_mmask8, \_\_m256i a, \_\_m256i b);

VPTESTNMQ \_\_mmask8 \_\_mm\_testn\_epi64\_mask(\_\_m128i a, \_\_m128i b);

VPTESTNMQ \_\_mmask8 \_mm\_mask\_testn\_epi64\_mask(\_\_mmask8, \_\_m128i a, \_\_m128i b);

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

VPTESTNMD/VPTESTNMQ: See Exceptions Type E4.

VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb.

## VRANGEPD—Range Restriction Calculation For Packed Pairs of Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 50 /r ib VRANGEPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL 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 VRANGEPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL 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 VRANGEPD zmm1 {k1}{z}, zmm2, 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 FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

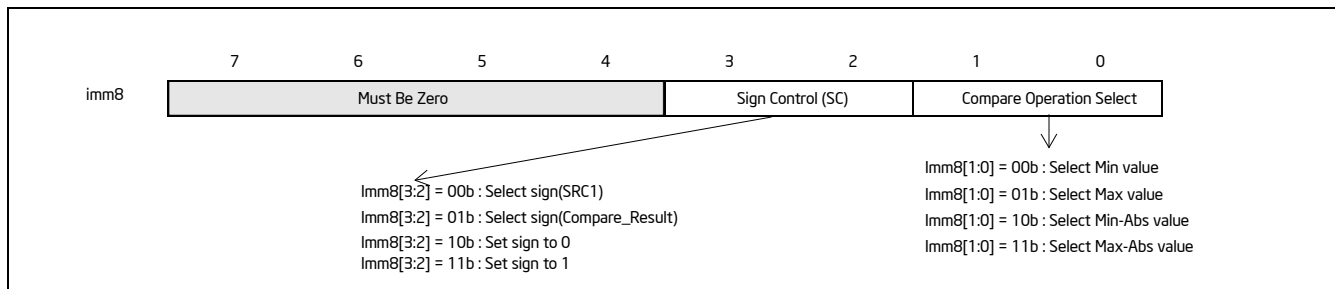


Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS

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 5-10. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-10.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-12.

**Table 5-10. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]**

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 5-11. 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 5-12. Comparison Result of Equal-Magnitude Input Cases for MIN\_ABS and MAX\_ABS, ( $|a| = |b|$ ,  $a > 0$ ,  $b < 0$ )**

MIN_ABS ( $ a  =  b $ , $a > 0$ , $b < 0$ )			MAX_ABS ( $ a  =  b $ , $a > 0$ , $b < 0$ )		
Src1	Src2	Result	Src1	Src2	Result
a	b	b	a	b	a
b	a	b	b	a	a

**Operation**

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```

{
  // Check if SNAN and report IE, see also Table 5-10
  IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
  IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

  Src1.exp ← SRC1[62:52];
  Src1.fraction ← SRC1[51:0];
  IF ((Src1.exp = 0) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
  FI;

  Src2.exp ← SRC2[62:52];
  Src2.fraction ← SRC2[51:0];
  IF ((Src2.exp = 0) and (Src2.fraction !=0)) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
  FI;

  IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
  ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
  ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-11
  ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-12
  ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
    01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
    10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
    11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
    ESAC;
  FI;

  Case(SignSelCtl[1:0])
  00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
  01: dest ← TMP[63:0];// Preserve sign of compare result
  10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
  11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
  ESAC;
  RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```



**VRANGEPD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

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

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

THEN DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);

ELSE DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);

FI;

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+63:i] = 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 1023$ .

```
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 Exceptions Type E2.

## VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 50 /r ib VRANGEPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL 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 VRANGEPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL 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 VRANGEPS zmm1 {k1}{z}, zmm2, 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 FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-10. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-10.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPS/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-12.

**Operation**

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```
{
  // Check if SNAN and report IE, see also Table 5-10
  IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
  IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

  Src1.exp ← SRC1[30:23];
  Src1.fraction ← SRC1[22:0];
  IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
  FI;
  Src2.exp ← SRC2[30:23];
  Src2.fraction ← SRC2[22:0];
  IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
  FI;

  IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
  ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
  ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-11
  ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-12
  ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
    01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
    10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
    11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
    ESAC;
  FI;
  Case(SignSelCtl[1:0])
  00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
  01: dest ← TMP[31:0];// Preserve sign of compare result
  10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
  11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
  ESAC;
  RETURN dest[31:0];
}
```

CmpOpCtl[1:0]= imm8[1:0];

SignSelCtl[1:0]=imm8[3:2];

**VRANGEPS**

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

FOR j ← 0 TO KL-1

i ← j \* 32

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

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

THEN DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);

ELSE DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[i+31:i], CmpOpCtl[1:0], SignSelCtl[1:0]);

FI;

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] = 0

FI;

FI;

ENDFOR;

DEST[MAXVL-1:VL] ← 0

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 150$ .

```
VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;
```

Where:

zmm\_dst is the destination operand.

zmm\_src is the input operand to compare against  $\pm 150$ .

zmm\_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of src1.sign.

In case  $|zmm\_src| < 150$ , then its value will be written into zmm\_dst. Otherwise, the value stored in zmm\_dst will get the value of 150 (received on zmm\_150).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm\_src. So, even in the case of  $|zmm\_src| \geq 150$ , the selected sign of SRC1 is kept.

Thus, if  $zmm\_src < -150$ , the result of VRANGEPS will be the minimal value of -150 while if  $zmm\_src > +150$ , the result of VRANGE will be the maximal value of +150.

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

```

VRANGEPS __m512_mm512_range_ps (__m512 a, __m512 b, int imm);
VRANGEPS __m512_mm512_range_round_ps (__m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512_mm512_mask_range_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512_mm512_mask_range_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512_mm512_maskz_range_ps (__mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512_mm512_maskz_range_round_ps (__mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256_mm256_range_ps (__m256 a, __m256 b, int imm);
VRANGEPS __m256_mm256_mask_range_ps (__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256_mm256_maskz_range_ps (__mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128_mm_range_ps (__m128 a, __m128 b, int imm);
VRANGEPS __m128_mm_mask_range_ps (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128_mm_maskz_range_ps (__mmask8 k, __m128 a, __m128 b, int imm);

```

**SIMD Floating-Point Exceptions**

Invalid, Denormal

**Other Exceptions**

See Exceptions Type E2.

## VRANGESD—Range Restriction Calculation From a pair of Scalar Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W1 51 /r VRANGESD xmm1 {k1}{z}, 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)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

### Description

This instruction calculates a range operation output from two input double-precision FP values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-10. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-10.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGESD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-12.

**Operation**

RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])

```

{
  // Check if SNAN and report IE, see also Table 5-10
  IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
  IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

  Src1.exp ← SRC1[62:52];
  Src1.fraction ← SRC1[51:0];
  IF ((Src1.exp = 0) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
    IF DAZ THEN Src1.fraction ← 0;
    ELSE IF (SRC2 <> QNAN) Set DE; FI;
  FI;

  Src2.exp ← SRC2[62:52];
  Src2.fraction ← SRC2[51:0];
  IF ((Src2.exp = 0) and (Src2.fraction !=0)) THEN// Src2 is a denormal number
    IF DAZ THEN Src2.fraction ← 0;
    ELSE IF (SRC1 <> QNAN) Set DE; FI;
  FI;

  IF (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
  ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
  ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-11
  ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-12
  ELSE
    Case(CmpOpCtl[1:0])
    00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
    01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
    10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
    11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
    ESAC;
  FI;

  Case(SignSelCtl[1:0])
  00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
  01: dest ← TMP[63:0];// Preserve sign of compare result
  10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
  11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
  ESAC;
  RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```



**VRANGESD**

```

IF k1[0] OR *no writemask*
  THEN DEST[63:0] ← RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
ELSE
  IF *merging-masking*           ; merging-masking
    THEN *DEST[63:0] remains unchanged*
  ELSE                             ; zeroing-masking
    DEST[63:0] = 0
  FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

```

The following example describes a common usage of this instruction for checking that the input operand is bounded between  $\pm 1023$ .

```
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 Exceptions Type E3.

## VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W0 51 /r VRANGESS xmm1 {k1}{z}, 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)	NA

### Description

This instruction calculates a range operation output from two input single-precision FP values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

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 of more input value is NAN is listed in Table 5-10. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-10.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN\_ABS/MAX\_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-11.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN\_ABS or MAX\_ABS comparison operation with result listed in Table 5-12.

**Operation**

```

RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-10
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[30:23];
    Src1.fraction ← SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp ← SRC2[30:23];
    Src2.fraction ← SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-11
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-12
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest ← TMP[31:0];// Preserve sign of compare result
    10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];

```

**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 Exceptions Type E3.

## VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4C /r VRCP14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VRCP14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VRCP14PD zmm1 {k1}{z}, 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)	NA	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand. The maximum relative error for this approximation is less than  $2^{-14}$ .

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-13. VRCP14PD/VRCP14SD Special Cases**

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-vrcp14-vrsqrt14-vrcp28-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 Exceptions Type E4.

## VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W1 4D /r VRCP14SD xmm1 {k1}{z}, xmm2, 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)	NA

### Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double-precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-13 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at:

<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 Exceptions Type E5.



## VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4C /r VRCP14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4C /r VRCP14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4C /r VRCP14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

### Instruction Operand Encoding

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

### Description

This instruction performs a SIMD computation of the approximate reciprocals of the packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than  $2^{-14}$ .

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

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

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

**Table 5-14. VRCP14PS/VRCP14SS Special Cases**

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*
$X < -2^{126}$	-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-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 Exceptions Type E4.

## VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W0 4D /r VRCP14SS xmm1 {k1}{z}, xmm2, 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)	NA

### 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 5-14 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 Exceptions Type E5.

## VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 56 /r ib VREDUCEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL 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 VREDUCEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL 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 VREDUCEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, 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	NA

### Description

Perform reduction transformation of the packed binary encoded double-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , 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.

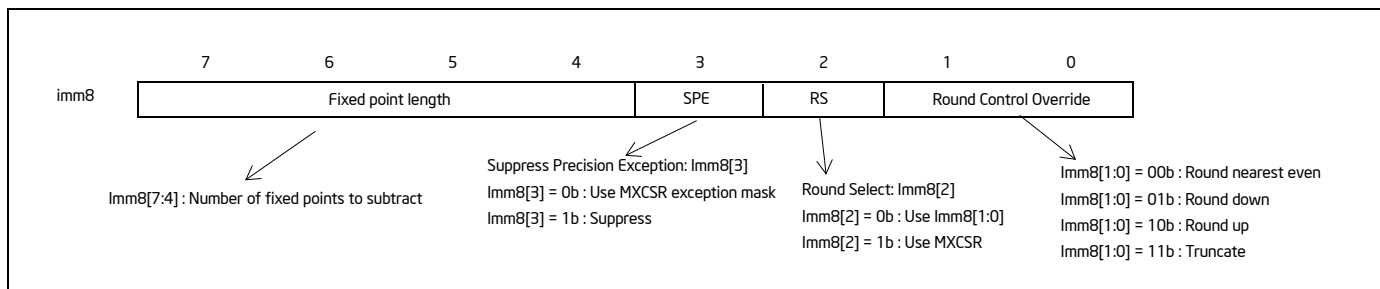


Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-15.

**Table 5-15. 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 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 ← 0; // 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 Exceptions Type E2, additionally

#UD                      If EVEX.vvvv != 1111B.

### VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W1 57 VREDUCESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8/r	A	V/V	AVX512D 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)	NA

#### Description

Perform a reduction transformation of the binary encoded double-precision FP value in the low qword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , 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.

The operation is write masked.

Handling of special case of input values are listed in Table 5-15.

#### Operation

ReduceArgumentDP(SRC[63:0], imm8[7:0])

```

{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0]; // Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0; // Suppress Precision Exception
    TMP[63:0] ← 2-M * {ROUND(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 Exceptions Type E3.

## VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 56 /r ib VREDUCEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL 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 VREDUCEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL 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 VREDUCEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, 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	NA

### Description

Perform reduction transformation of the packed binary encoded single-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and bi-ased 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

$$\text{Then if RC} = \text{RNE: } 0 \leq |\text{Reduced Result}| \leq 2^{p-M-1}$$

$$\text{Then if RC} \neq \text{RNE: } 0 \leq |\text{Reduced Result}| < 2^{p-M}$$

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

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

Handling of special case of input values are listed in Table 5-15.

**Operation**

```

ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0]; // Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0; // 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 Exceptions Type E2, additionally

#UD If EVEX.vvvv != 1111B.

## VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W0 57 /r /ib VREDUCESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, 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)	NA

### Description

Perform a reduction transformation of the binary encoded single-precision FP value in the low dword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

$$\text{dest} = \text{src} - (\text{ROUND}(2^M * \text{src})) * 2^{-M};$$

where "Round()" treats "src", "2<sup>M</sup>", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering  $\text{src} = 2^p * \text{man}_2$ , 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 5-15.

### Operation

ReduceArgumentSP(SRC[31:0], imm8[7:0])

```
{
  // Check for NaN
  IF (SRC [31:0] = NAN) THEN
    RETURN (Convert SRC[31:0] to QNaN); FI
  M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
  RC ← imm8[1:0]; // Round Control for ROUND() operation
  RC source ← imm[2];
  SPE ← 0; // 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 Exceptions Type E3.

## VRNDSCALEPD—Round Packed Float64 Values To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 09 /r ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL 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 VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL 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 VRNDSCALEPD zmm1 {k1}{z}, 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	NA

### 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.

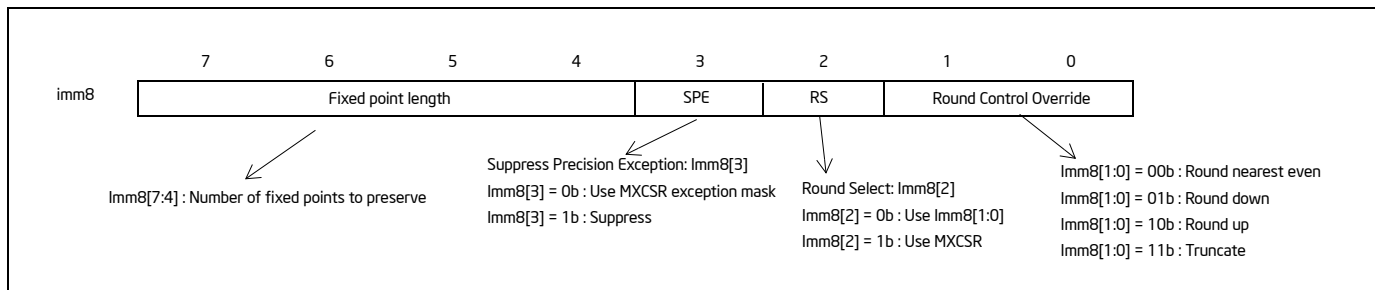


Figure 5-29. Imm8 Controls for VRNDSCALEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-16.

Table 5-16. 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 Exceptions Type E2.

## VRNDSCALESD—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.OF3A.W1 0B /r ib VRNDSCALESD xmm1 {k1}{z}, xmm2, 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

$$\begin{aligned} \text{ROUND}(x) &= 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \\ M &= \text{imm}[7:4]; \end{aligned}$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

$$\begin{aligned} \text{ROUND}(x) &= \text{Round\_to\_INT}(x, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \end{aligned}$$

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-16.

**Operation**

```

RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4] ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(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 Exceptions Type E3.

## VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 08 /r ib VRNDSCALEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed single-precision floating point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W0 08 /r ib VRNDSCALEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed single-precision floating point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W0 08 /r ib VRNDSCALEPS zmm1 {k1}{z}, 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	NA

### 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

$$\begin{aligned} \text{ROUND}(x) &= 2^{-M} * \text{Round\_to\_INT}(x * 2^M, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \\ M &= \text{imm}[7:4]; \end{aligned}$$

The operation of  $x * 2^M$  is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

$$\begin{aligned} \text{ROUND}(x) &= \text{Round\_to\_INT}(x, \text{round\_ctrl}), \\ \text{round\_ctrl} &= \text{imm}[3:0]; \end{aligned}$$

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.  
Handling of special case of input values are listed in Table 5-16.

### Operation

```

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR.RC ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0] ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4] ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] ← round_to_nearest_even_integer(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 Exceptions Type E2.

## VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, 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)	NA

### 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 5-16.

**Operation**

```

RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
  if (imm8[2] = 1)
    rounding_direction ← MXCSR:RC    ; get round control from MXCSR
  else
    rounding_direction ← imm8[1:0]    ; get round control from imm8[1:0]
  FI
  M ← imm8[7:4]                      ; get the scaling factor

  case (rounding_direction)
  00: TMP[31:0] ← round_to_nearest_even_integer(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 Exceptions Type E3.



## VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4E /r VRSQRT14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL 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 VRSQRT14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL 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 VRSQRT14PD zmm1 {k1}{z}, 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)	NA	NA

### 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 5-17. 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 Exceptions Type E4.

## VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W1 4F /r VRSQRT14SD xmm1 {k1}{z}, 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)	NA

### 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 5-18. 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 Exceptions Type E5.

## VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4E /r VRSQRT14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL 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 VRSQRT14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL 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 VRSQRT14PS zmm1 {k1}{z}, 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)	NA	NA

### 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 5-19. 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 Exceptions Type 4.

## VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W0 4F /r VRSQRT14SS xmm1 {k1}{z}, 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	ModRM:r/m (r)	NA

### Description

Computes of the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than  $2^{-14}$ . The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an  $\infty$  with the sign of the source value is returned. When the source operand is an  $\infty$ , zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

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]  $\leftarrow$  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]  $\leftarrow$  0

FI;

FI;

DEST[127:32]  $\leftarrow$  SRC1[127:32]

DEST[MAXVL-1:128]  $\leftarrow$  0

Table 5-20. 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\_rsqr14\_ss( \_\_m128 a, \_\_m128 b);

VRSQRT14SS \_\_m128 \_\_mm\_mask\_rsqr14\_ss( \_\_m128 s, \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

VRSQRT14SS \_\_m128 \_\_mm\_maskz\_rsqr14\_ss( \_\_mmask8 k, \_\_m128 a, \_\_m128 b);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

See Exceptions Type E5.



## VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 2C /r VSCALEFPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL 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 VSCALEFPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL 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 VSCALEFPD zmm1 {k1}{z}, zmm2, 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)	NA

### Description

Performs a floating-point scale of the packed double-precision floating-point values in the first source operand by multiplying it by 2 power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

$$zmm1 := zmm2 * 2^{\text{floor}(zmm3)}$$

Floor(zmm3) means maximum integer value  $\leq$  zmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

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

Handling of special-case input values are listed in Table 5-21 and Table 5-22.

**Table 5-21. 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 5-22. 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_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask* THEN
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
      THEN DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[63:0]);
    ELSE DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[i+63:i]);
    FI;
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[i+63:i] remains unchanged*
    ELSE ; zeroing-masking
      DEST[i+63:i] ← 0
    FI
  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 Exceptions Type E2.

## VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W1 2D /r VSCALEFSD xmm1 {k1}{z}, xmm2, 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)	NA

### Description

Performs a floating-point scale of the packed double-precision floating-point value in the first source operand by multiplying it by 2 power of the double-precision floating-point value in second source operand.

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{Floor}(\text{xmm3})}$$

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 5-21 and Table 5-22.

### Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 64 bits floating-point value */
    DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

**VSCALEFSD (EVEX encoded version)**

```

IF (EVEX.b= 1) and SRC2 *is a register*
  THEN
    SET_RM(EVEX.RC);
  ELSE
    SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
  THEN DEST[63:0] ← SCALE(SRC1[63:0], SRC2[63:0])
  ELSE
    IF *merging-masking* ; merging-masking
      THEN *DEST[63:0] remains unchanged*
    ELSE ; zeroing-masking
      DEST[63:0] ← 0
    FI
  FI;
DEST[127:64] ← SRC1[127:64]
DEST[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 Exceptions Type E3.

## VSCALEFPS—Scale Packed Float32 Values With Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 2C /r VSCALEFPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL 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 VSCALEFPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL 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 VSCALEFPS zmm1 {k1}{z}, zmm2, 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)	NA

### Description

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

$$zmm1 := zmm2 * 2^{\text{floor}(zmm3)}$$

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 5-21 and Table 5-23.

**Table 5-23. 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\_RM(EVEX.RC);

ELSE

SET\_RM(MXCSR.RM);

FI;

FOR j ← 0 TO KL-1

i ← j \* 32

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

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

THEN DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[31:0]);

ELSE DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[i+31:i]);

FI;

ELSE

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

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

ELSE ; zeroing-masking

DEST[i+31:i] ← 0

FI

FI;

ENDFOR

DEST[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 Exceptions Type E2.



## VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

Opcode/ Instruction	Op/ En	64/32 bitMode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F38.W0 2D /r VSCALEFSS xmm1 {k1}{z}, xmm2, 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)	NA

### Description

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 power of the float32 value in second source operand.

The equation of this operation is given by:

$$\text{xmm1} := \text{xmm2} * 2^{\text{floor}(\text{xmm3})}$$

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 5-21 and Table 5-23.

**Operation**

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
    TMP_SRC2 ← SRC2
    TMP_SRC1 ← SRC1
    IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
    IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
    /* SRC2 is a 32 bits floating-point value */
    DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFSS (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking* ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE ; zeroing-masking
                DEST[31:0] ← 0
        FI
    FI;
DEST[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 Exceptions Type E3.

## VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A2 /vsib 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 VSCATTERDPD vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A2 /vsib 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 VSCATTERQPS vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A3 /vsib 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 VSCATTERQPD vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A3 /vsib 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, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

#### Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address `BASE_ADDR` and index vector `VINDEX`, with scale `SCALE`. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, `EFLAG.RF` is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.

- If two or more destination indices completely overlap, the “earlier” write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special  $\text{disp8} * N$  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[j+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[i+31:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

**VSCATTERQPD (EVEX encoded versions)**

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

```

FOR j ← 0 TO KL-1
  i ← j * 64
  IF k1[j] OR *no writemask*
    THEN MEM[BASE_ADDR + (VINDEK[i+63:i]) * SCALE + DISP] ←
      SRC[i+63:i]
      k1[j] ← 0
  FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0

```

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

```

VSCATTERDPD void __mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERDPD void __mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void __mm512_i32scatter_ps(void * base, __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 Exceptions Type E12.

## VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-bit Granularity

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F3A.W0 23 /r ib VSHUFF32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL 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 VSHUFF32x4 zmm1{k1}{z}, zmm2, 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 VSHUFF64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL 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 VSHUFF64x2 zmm1{k1}{z}, zmm2, 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 VSHUFI32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL 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 VSHUFI32x4 zmm1{k1}{z}, zmm2, 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 VSHUFI64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL 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 VSHUFI64x2 zmm1{k1}{z}, zmm2, 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)	NA

### 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[j+31:i] ← TMP_DEST[j+31:i]
  ELSE
    IF *merging-masking*           ; merging-masking
      THEN *DEST[j+31:i] remains unchanged*
    ELSE *zeroing-masking*         ; zeroing-masking
      THEN DEST[j+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[j+63:i] ← SRC2[63:0]
    ELSE TMP_SRC2[j+63:i] ← SRC2[j+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[j+63:i] ← TMP_DEST[j+63:i]
    ELSE
      IF *merging-masking*           ; merging-masking
        THEN *DEST[j+63:i] remains unchanged*
      ELSE *zeroing-masking*         ; zeroing-masking
        THEN DEST[j+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 Exceptions Type E4NF.

#UD If EVEX.L'L = 0 for VSHUFF32x4/VSHUFF64x2.

## VTESTPD/VTESTPS—Packed Bit Test

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0E /r VTESTPS <i>xmm1, xmm2/m128</i>	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 VTESTPS <i>ymm1, ymm2/m256</i>	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 VTESTPD <i>xmm1, xmm2/m128</i>	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 VTESTPD <i>ymm1, ymm2/m256</i>	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)	NA	NA

### 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 Exceptions Type 4; additionally

#UD	If VEX.vvvv ≠ 1111B.
	If VEX.W = 1 for VTESTPS or VTESTPD.

## VZEROALL—Zero All YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.OF.WIG 77 VZEROALL	Z0	V/V	AVX	Zero all YMM registers.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

The instruction zeros contents of all XMM or YMM registers.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

### Operation

`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. On a machine supporting Intel AVX but not Intel AVX-512, the width is "MAXVL".

### VZEROALL (VEX.256 encoded version)

IF (64-bit mode)

limit  $\leftarrow$  15

ELSE

limit  $\leftarrow$  7

FOR i in 0 .. limit:

`simd_reg_file[i][MAXVL-1:0]  $\leftarrow$  0`

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

VZEROALL:     `_mm256_zeroall()`

### SIMD Floating-Point Exceptions

None.

### Other Exceptions

See Exceptions Type 8.

## VZEROUPPER—Zero Upper Bits of YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.0F.WIG 77 VZEROUPPER	Z0	V/V	AVX	Zero upper 128 bits of all YMM registers.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

The instruction zeros the bits in position 128 and higher of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

### Operation

`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. On a machine supporting Intel AVX but not Intel AVX-512, the width is "MAXVL".

### 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 Exceptions Type 8.



**WAIT/FWAIT—Wait**

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
9B	WAIT	Z0	Valid	Valid	Check pending unmasked floating-point exceptions.
9B	FWAIT	Z0	Valid	Valid	Check pending unmasked floating-point exceptions.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

**Operation**

CheckForPendingUnmaskedFloatingPointExceptions;

**FPU Flags Affected**

The C0, C1, C2, and C3 flags are undefined.

**Floating-Point Exceptions**

None.

**Protected Mode Exceptions**

#NM If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.

#UD If the LOCK prefix is used.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 09	WBINVD	Z0	Valid	Valid	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Writes back all modified cache lines in the processor's internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction. This instruction's operation is the same in non-64-bit modes and 64-bit mode.

### IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

### Operation

```
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)
```

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

```
WBINVD void _wbinvd(void);
```

### Flags Affected

None.

### Protected Mode Exceptions

```
#GP(0)      If the current privilege level is not 0.
#UD        If the LOCK prefix is used.
```

**Real-Address Mode Exceptions**

#UD                      If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)                  WBINVD cannot be executed at the virtual-8086 mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

**WRFSBASE/WRGSBASE—Write FS/GS Segment Base**

Opcode/ Instruction	Op/ En	64/32- bit Mode	CPUID Fea- ture Flag	Description
F3 OF AE /2 WRFSBASE <i>r32</i>	M	V/I	FSGSBASE	Load the FS base address with the 32-bit value in the source register.
F3 REX.W OF AE /2 WRFSBASE <i>r64</i>	M	V/I	FSGSBASE	Load the FS base address with the 64-bit value in the source register.
F3 OF AE /3 WRGSBASE <i>r32</i>	M	V/I	FSGSBASE	Load the GS base address with the 32-bit value in the source register.
F3 REX.W OF AE /3 WRGSBASE <i>r64</i>	M	V/I	FSGSBASE	Load the GS base address with the 64-bit value in the source register.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

**Description**

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field.

The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

**Operation**

FS/GS segment base address ← SRC;

**Flags Affected**

None

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

```
WRFSBASE:    void _writefsbase_u32( unsigned int );
WRFSBASE:    _writefsbase_u64( unsigned __int64 );
WRGSBASE:    void _writegsbase_u32( unsigned int );
WRGSBASE:    _writegsbase_u64( unsigned __int64 );
```

**Protected Mode Exceptions**

#UD The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

**Real-Address Mode Exceptions**

#UD The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

**Virtual-8086 Mode Exceptions**

#UD The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

**Compatibility Mode Exceptions**

#UD The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.

## 64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.FSGSBASE[bit 16] = 0. If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0
#GP(0)	If the source register contains a non-canonical address.

## WRMSR—Write to Model Specific Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF 30	WRMSR	Z0	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see “Translation Lookaside Buffers (TLBs)” in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 2, “Model-Specific Registers (MSRs)” of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 4*, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see “Serializing Instructions” in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*). Note that WRMSR to the IA32\_TSC\_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

### IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

### Operation

MSR[ECX] ← EDX:EAX;

### Flags Affected

None.

**Protected Mode Exceptions**

#GP(0)	<p>If the current privilege level is not 0.</p> <p>If the value in ECX specifies a reserved or unimplemented MSR address.</p> <p>If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.</p> <p>If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.</p>
#UD	If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	<p>If the value in ECX specifies a reserved or unimplemented MSR address.</p> <p>If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.</p> <p>If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.</p>
#UD	If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0)	The WRMSR instruction is not recognized in virtual-8086 mode.
--------	---

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

**WRPKRU—Write Data to User Page Key Register**

Opcode*	Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EF	WRPKRU	Z0	V/V	OSPKE	Writes EAX into PKRU.

**Instruction Operand Encoding**

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

**Description**

Writes the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX and RAX are ignored.

**Operation**

```
IF (ECX = 0 AND EDX = 0)
    THEN PKRU ← EAX;
    ELSE #GP(0);
FI;
```

**Flags Affected**

None.

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

```
WRPKRU:    void _wrpkru(uint32_t);
```

**Protected Mode Exceptions**

#GP(0)            If ECX ≠ 0.  
                   If EDX ≠ 0.  
 #UD              If the LOCK prefix is used.  
                   If CR4.PKE = 0.

**Real-Address Mode Exceptions**

Same exceptions as in protected mode.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.



## XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE <sup>1</sup>	A hint used with an “XACQUIRE-enabled” instruction to start lock elision on the instruction memory operand address.
F3 XRELEASE	V/V	HLE	A hint used with an “XRELEASE-enabled” instruction to end lock elision on the instruction memory operand address.

### NOTES:

- Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

### Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The “MOV mem, reg” (Opcode 88H/89H) and “MOV mem, imm” (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.

**Intel 64 and IA-32 Compatibility**

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to

- REPNE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to

- REP/REPE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

**Operation****XACQUIRE**

IF XACQUIRE-enabled instruction

THEN

IF (HLE\_NEST\_COUNT < MAX\_HLE\_NEST\_COUNT) THEN

HLE\_NEST\_COUNT++

IF (HLE\_NEST\_COUNT = 1) THEN

HLE\_ACTIVE ← 1

IF 64-bit mode

THEN

restartRIP ← instruction pointer of the XACQUIRE-enabled instruction

ELSE

restartEIP ← instruction pointer of the XACQUIRE-enabled instruction

FI;

Enter HLE Execution (\* record register state, start tracking memory state \*)

FI; (\* HLE\_NEST\_COUNT = 1 \*)

IF ElisionBufferAvailable

THEN

Allocate elision buffer

Record address and data for forwarding and commit checking

Perform elision

ELSE

Perform lock acquire operation transactionally but without elision

FI;

ELSE (\* HLE\_NEST\_COUNT = MAX\_HLE\_NEST\_COUNT \*)

GOTO HLE\_ABORT\_PROCESSING

FI;

ELSE

Treat instruction as non-XACQUIRE F2H prefixed legacy instruction

FI;

**XRELEASE**

```

IF XRELEASE-enabled instruction
  THEN
    IF (HLE_NEST_COUNT > 0)
      THEN
        HLE_NEST_COUNT--
        IF lock address matches in elision buffer THEN
          IF lock satisfies address and value requirements THEN
            Deallocate elision buffer
          ELSE
            GOTO HLE_ABORT_PROCESSING
          FI;
        FI;
      IF (HLE_NEST_COUNT = 0)
        THEN
          IF NoAllocatedElisionBuffer
            THEN
              Try to commit transactional execution
              IF fail to commit transactional execution
                THEN
                  GOTO HLE_ABORT_PROCESSING;
                ELSE (* commit success *)
                  HLE_ACTIVE ← 0
                FI;
            ELSE
              GOTO HLE_ABORT_PROCESSING
            FI;
          FI;
        FI; (* HLE_NEST_COUNT > 0 *)
      ELSE
        Treat instruction as non-XRELEASE F3H prefixed legacy instruction
    FI;

```

(\* For any HLE abort condition encountered during HLE execution \*)

```

HLE_ABORT_PROCESSING:
  HLE_ACTIVE ← 0
  HLE_NEST_COUNT ← 0
  Restore architectural register state
  Discard memory updates performed in transaction
  Free any allocated lock elision buffers
  IF 64-bit mode
    THEN
      RIP ← restartRIP
    ELSE
      EIP ← restartEIP
  FI;
  Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END

```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

#GP(0) If the use of prefix causes instruction length to exceed 15 bytes.

## XABORT – Transactional Abort

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
C6 F8 ib XABORT imm8	A	V/V	RTM	Causes an RTM abort if in RTM execution

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	imm8	NA	NA	NA

### Description

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

### Operation

#### XABORT

```
IF RTM_ACTIVE = 0
  THEN
    Treat as NOP;
  ELSE
    GOTO RTM_ABORT_PROCESSING;
FI;
```

(\* For any RTM abort condition encountered during RTM execution \*)

```
RTM_ABORT_PROCESSING:
  Restore architectural register state;
  Discard memory updates performed in transaction;
  Update EAX with status and XABORT argument;
  RTM_NEST_COUNT ← 0;
  RTM_ACTIVE ← 0;
  IF 64-bit Mode
    THEN
      RIP ← fallbackRIP;
    ELSE
      EIP ← fallbackEIP;
  FI;
END
```

### Flags Affected

None

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

XABORT: `void _xabort(unsigned int);`

### SIMD Floating-Point Exceptions

None

**Other Exceptions**

#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.  
If LOCK prefix is used.

## XADD—Exchange and Add

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
OF C0 /r	XADD r/m8, r8	MR	Valid	Valid	Exchange r8 and r/m8; load sum into r/m8.
REX + OF C0 /r	XADD r/m8*, r8*	MR	Valid	N.E.	Exchange r8 and r/m8; load sum into r/m8.
OF C1 /r	XADD r/m16, r16	MR	Valid	Valid	Exchange r16 and r/m16; load sum into r/m16.
OF C1 /r	XADD r/m32, r32	MR	Valid	Valid	Exchange r32 and r/m32; load sum into r/m32.
REX.W + OF C1 /r	XADD r/m64, r64	MR	Valid	N.E.	Exchange r64 and r/m64; load sum into r/m64.

### NOTES:

\* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r, w)	NA	NA

### Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

### IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

### Operation

```
TEMP ← SRC + DEST;
SRC ← DEST;
DEST ← TEMP;
```

### Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

### Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Real-Address Mode Exceptions

- #GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS If a memory operand effective address is outside the SS segment limit.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

### Virtual-8086 Mode Exceptions

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made.
- #UD If the LOCK prefix is used but the destination is not a memory operand.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

- #SS(0) If a memory address referencing the SS segment is in a non-canonical form.
- #GP(0) If the memory address is in a non-canonical form.
- #PF(fault-code) If a page fault occurs.
- #AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
- #UD If the LOCK prefix is used but the destination is not a memory operand.



## XBEGIN – Transactional Begin

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
C7 F8 XBEGIN rel16	A	V/V	RTM	Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.
C7 F8 XBEGIN rel32	A	V/V	RTM	Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	Offset	NA	NA	NA

### Description

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort.

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

### Operation

#### XBEGIN

```

IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
  THEN
    RTM_NEST_COUNT++
    IF RTM_NEST_COUNT = 1 THEN
      IF 64-bit Mode
        THEN
          fallbackRIP ← RIP + SignExtend64(IMM)
                          (* RIP is instruction following XBEGIN instruction *)
        ELSE
          fallbackEIP ← EIP + SignExtend32(IMM)
                          (* EIP is instruction following XBEGIN instruction *)
      FI;

      IF (64-bit mode)
        THEN IF (fallbackRIP is not canonical)
              THEN #GP(0)
              FI;
          ELSE IF (fallbackEIP outside code segment limit)
              THEN #GP(0)
              FI;
          FI;

      RTM_ACTIVE ← 1
      Enter RTM Execution (* record register state, start tracking memory state*)
      FI; (* RTM_NEST_COUNT = 1 *)

```

```

ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
    GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END

```

**Flags Affected**

None

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

XBEGIN: unsigned int \_xbegin( void );

**SIMD Floating-Point Exceptions**

None

**Protected Mode Exceptions**

#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.  
If LOCK prefix is used.

#GP(0) If the fallback address is outside the CS segment.

**Real-Address Mode Exceptions**

#GP(0) If the fallback address is outside the address space 0000H and FFFFH.

#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.  
If LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

#GP(0) If the fallback address is outside the address space 0000H and FFFFH.

#UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.  
If LOCK prefix is used.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

## 64-bit Mode Exceptions

- #UD CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.  
If LOCK prefix is used.
- #GP(0) If the fallback address is non-canonical.

## XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
90+ <i>rw</i>	XCHG AX, <i>r16</i>	0	Valid	Valid	Exchange <i>r16</i> with AX.
90+ <i>rw</i>	XCHG <i>r16</i> , AX	0	Valid	Valid	Exchange AX with <i>r16</i> .
90+ <i>rd</i>	XCHG EAX, <i>r32</i>	0	Valid	Valid	Exchange <i>r32</i> with EAX.
REX.W + 90+ <i>rd</i>	XCHG RAX, <i>r64</i>	0	Valid	N.E.	Exchange <i>r64</i> with RAX.
90+ <i>rd</i>	XCHG <i>r32</i> , EAX	0	Valid	Valid	Exchange EAX with <i>r32</i> .
REX.W + 90+ <i>rd</i>	XCHG <i>r64</i> , RAX	0	Valid	N.E.	Exchange RAX with <i>r64</i> .
86 <i>lr</i>	XCHG <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	Exchange <i>r8</i> (byte register) with byte from <i>r/m8</i> .
REX + 86 <i>lr</i>	XCHG <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	Exchange <i>r8</i> (byte register) with byte from <i>r/m8</i> .
86 <i>lr</i>	XCHG <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	Exchange byte from <i>r/m8</i> with <i>r8</i> (byte register).
REX + 86 <i>lr</i>	XCHG <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	Exchange byte from <i>r/m8</i> with <i>r8</i> (byte register).
87 <i>lr</i>	XCHG <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	Exchange <i>r16</i> with word from <i>r/m16</i> .
87 <i>lr</i>	XCHG <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	Exchange word from <i>r/m16</i> with <i>r16</i> .
87 <i>lr</i>	XCHG <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	Exchange <i>r32</i> with doubleword from <i>r/m32</i> .
REX.W + 87 <i>lr</i>	XCHG <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	Exchange <i>r64</i> with quadword from <i>r/m64</i> .
87 <i>lr</i>	XCHG <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	Exchange doubleword from <i>r/m32</i> with <i>r32</i> .
REX.W + 87 <i>lr</i>	XCHG <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	Exchange quadword from <i>r/m64</i> with <i>r64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
0	AX/EAX/RAX ( <i>r</i> , <i>w</i> )	opcode + <i>rd</i> ( <i>r</i> , <i>w</i> )	NA	NA
0	opcode + <i>rd</i> ( <i>r</i> , <i>w</i> )	AX/EAX/RAX ( <i>r</i> , <i>w</i> )	NA	NA
MR	ModRM: <i>r/m</i> ( <i>r</i> , <i>w</i> )	ModRM:reg ( <i>r</i> )	NA	NA
RM	ModRM:reg ( <i>w</i> )	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA

### Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

**NOTE**

XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

**Operation**

TEMP ← DEST;  
 DEST ← SRC;  
 SRC ← TEMP;

**Flags Affected**

None.

**Protected Mode Exceptions**

#GP(0)	If either operand is in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Real-Address Mode Exceptions**

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Virtual-8086 Mode Exceptions**

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## XEND – Transactional End

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
NP OF 01 D5 XEND	A	V/V	RTM	Specifies the end of an RTM code region.

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

### Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

### Operation

#### XEND

```

IF (RTM_ACTIVE = 0) THEN
    SIGNAL #GP
ELSE
    RTM_NEST_COUNT--
    IF (RTM_NEST_COUNT = 0) THEN
        Try to commit transaction
        IF fail to commit transactional execution
            THEN
                GOTO RTM_ABORT_PROCESSING;
            ELSE (* commit success *)
                RTM_ACTIVE ← 0
    FI;
FI;
FI;

```

(\* For any RTM abort condition encountered during RTM execution \*)

```

RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END

```

**Flags Affected**

None

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

XEND:       void\_xend( void );

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0. If LOCK prefix is used.
#GP(0)	If RTM_ACTIVE = 0.

## XGETBV—Get Value of Extended Control Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP 0F 01 D0	XGETBV	Z0	Valid	Valid	Reads an XCR specified by ECX into EDX:EAX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, “Managing State Using the XSAVE Feature Set,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Use of any other value for ECX results in a general-protection (#GP) exception.

### Operation

EDX:EAX ← XCR[ECX];

### Flags Affected

None.

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

XGETBV: `unsigned __int64 _xgetbv( unsigned int);`

### Protected Mode Exceptions

- #GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.  
If CR4.OSXSAVE[bit 18] = 0.  
If the LOCK prefix is used.

### Real-Address Mode Exceptions

- #GP(0) If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
- #UD If CPUID.01H:ECX.XSAVE[bit 26] = 0.  
If CR4.OSXSAVE[bit 18] = 0.  
If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.



**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
D7	XLAT <i>m8</i>	Z0	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	Z0	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
REX.W + D7	XLATB	Z0	Valid	N.E.	Set AL to memory byte [RBX + unsigned AL].

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the “explicit-operand” form and the “no-operand” form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operand form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operand form (XLATB) provides a “short form” of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table’s base address. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

```
IF AddressSize = 16
  THEN
    AL ← (DS:BX + ZeroExtend(AL));
  ELSE IF (AddressSize = 32)
    AL ← (DS:EBX + ZeroExtend(AL)); FI;
  ELSE (AddressSize = 64)
    AL ← (RBX + ZeroExtend(AL));
  FI;
```

### Flags Affected

None.

### Protected Mode Exceptions

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  
If the DS, ES, FS, or GS register contains a NULL segment selector.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF(fault-code) If a page fault occurs.

#UD If the LOCK prefix is used.

### Real-Address Mode Exceptions

#GP If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

#SS If a memory operand effective address is outside the SS segment limit.

#UD If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

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

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

#UD If the LOCK prefix is used.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0) If a memory address referencing the SS segment is in a non-canonical form.

#GP(0) If the memory address is in a non-canonical form.

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

#UD If the LOCK prefix is used.

## XOR—Logical Exclusive OR

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
34 <i>ib</i>	XOR AL, <i>imm8</i>	I	Valid	Valid	AL XOR <i>imm8</i> .
35 <i>iw</i>	XOR AX, <i>imm16</i>	I	Valid	Valid	AX XOR <i>imm16</i> .
35 <i>id</i>	XOR EAX, <i>imm32</i>	I	Valid	Valid	EAX XOR <i>imm32</i> .
REX.W + 35 <i>id</i>	XOR RAX, <i>imm32</i>	I	Valid	N.E.	RAX XOR <i>imm32</i> ( <i>sign-extended</i> ).
80 /6 <i>ib</i>	XOR <i>r/m8</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m8</i> XOR <i>imm8</i> .
REX + 80 /6 <i>ib</i>	XOR <i>r/m8*</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m8</i> XOR <i>imm8</i> .
81 /6 <i>iw</i>	XOR <i>r/m16</i> , <i>imm16</i>	MI	Valid	Valid	<i>r/m16</i> XOR <i>imm16</i> .
81 /6 <i>id</i>	XOR <i>r/m32</i> , <i>imm32</i>	MI	Valid	Valid	<i>r/m32</i> XOR <i>imm32</i> .
REX.W + 81 /6 <i>id</i>	XOR <i>r/m64</i> , <i>imm32</i>	MI	Valid	N.E.	<i>r/m64</i> XOR <i>imm32</i> ( <i>sign-extended</i> ).
83 /6 <i>ib</i>	XOR <i>r/m16</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m16</i> XOR <i>imm8</i> ( <i>sign-extended</i> ).
83 /6 <i>ib</i>	XOR <i>r/m32</i> , <i>imm8</i>	MI	Valid	Valid	<i>r/m32</i> XOR <i>imm8</i> ( <i>sign-extended</i> ).
REX.W + 83 /6 <i>ib</i>	XOR <i>r/m64</i> , <i>imm8</i>	MI	Valid	N.E.	<i>r/m64</i> XOR <i>imm8</i> ( <i>sign-extended</i> ).
30 /r	XOR <i>r/m8</i> , <i>r8</i>	MR	Valid	Valid	<i>r/m8</i> XOR <i>r8</i> .
REX + 30 /r	XOR <i>r/m8*</i> , <i>r8*</i>	MR	Valid	N.E.	<i>r/m8</i> XOR <i>r8</i> .
31 /r	XOR <i>r/m16</i> , <i>r16</i>	MR	Valid	Valid	<i>r/m16</i> XOR <i>r16</i> .
31 /r	XOR <i>r/m32</i> , <i>r32</i>	MR	Valid	Valid	<i>r/m32</i> XOR <i>r32</i> .
REX.W + 31 /r	XOR <i>r/m64</i> , <i>r64</i>	MR	Valid	N.E.	<i>r/m64</i> XOR <i>r64</i> .
32 /r	XOR <i>r8</i> , <i>r/m8</i>	RM	Valid	Valid	<i>r8</i> XOR <i>r/m8</i> .
REX + 32 /r	XOR <i>r8*</i> , <i>r/m8*</i>	RM	Valid	N.E.	<i>r8</i> XOR <i>r/m8</i> .
33 /r	XOR <i>r16</i> , <i>r/m16</i>	RM	Valid	Valid	<i>r16</i> XOR <i>r/m16</i> .
33 /r	XOR <i>r32</i> , <i>r/m32</i>	RM	Valid	Valid	<i>r32</i> XOR <i>r/m32</i> .
REX.W + 33 /r	XOR <i>r64</i> , <i>r/m64</i>	RM	Valid	N.E.	<i>r64</i> XOR <i>r/m64</i> .

### NOTES:

\* In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	<i>imm8/16/32</i>	NA	NA
MI	ModRM: <i>r/m</i> ( <i>r</i> , <i>w</i> )	<i>imm8/16/32</i>	NA	NA
MR	ModRM: <i>r/m</i> ( <i>r</i> , <i>w</i> )	ModRM: <i>reg</i> ( <i>r</i> )	NA	NA
RM	ModRM: <i>reg</i> ( <i>r</i> , <i>w</i> )	ModRM: <i>r/m</i> ( <i>r</i> )	NA	NA

### Description

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

### Operation

DEST ← DEST XOR SRC;

### Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

### Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If the DS, ES, FS, or GS register contains a NULL segment selector. If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

## XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57/r XORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical XOR of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.66.0F.W1 57 /r VXORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.256.66.0F.W1 57 /r VXORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.512.66.0F.W1 57 /r VXORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

### Instruction Operand Encoding

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

### Description

Performs a bitwise logical XOR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation****VXORPD (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 64

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

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

THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0];

ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i];

FI;

ELSE

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

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

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+63:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

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

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]

DEST[191:128] ← SRC1[191:128] BITWISE XOR SRC2[191:128]

DEST[255:192] ← SRC1[255:192] BITWISE XOR SRC2[255:192]

DEST[MAXVL-1:256] ← 0

**VXORPD (VEX.128 encoded version)**

DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]

DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]

DEST[MAXVL-1:128] ← 0

**XORPD (128-bit Legacy SSE version)**

DEST[63:0] ← DEST[63:0] BITWISE XOR SRC[63:0]

DEST[127:64] ← DEST[127:64] BITWISE XOR SRC[127:64]

DEST[MAXVL-1:128] (Unmodified)

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

VXORPD \_\_m512d \_\_mm512\_xor\_pd (\_\_m512d a, \_\_m512d b);

VXORPD \_\_m512d \_\_mm512\_mask\_xor\_pd (\_\_m512d a, \_\_mmask8 m, \_\_m512d b);

VXORPD \_\_m512d \_\_mm512\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m512d a);

VXORPD \_\_m256d \_\_mm256\_xor\_pd (\_\_m256d a, \_\_m256d b);

VXORPD \_\_m256d \_\_mm256\_mask\_xor\_pd (\_\_m256d a, \_\_mmask8 m, \_\_m256d b);

VXORPD \_\_m256d \_\_mm256\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m256d a);

XORPD \_\_m128d \_\_mm\_xor\_pd (\_\_m128d a, \_\_m128d b);

VXORPD \_\_m128d \_\_mm\_mask\_xor\_pd (\_\_m128d a, \_\_mmask8 m, \_\_m128d b);

VXORPD \_\_m128d \_\_mm\_maskz\_xor\_pd (\_\_mmask8 m, \_\_m128d a);

**SIMD Floating-Point Exceptions**

None

**Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.



## XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 57 /r XORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.128.0F.WIG 57 /r VXORPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.256.0F.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.128.0F.W0 57 /r VXORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.256.0F.W0 57 /r VXORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.512.0F.W0 57 /r VXORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

### Instruction Operand Encoding

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

#### Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

**Operation****VXORPS (EVEX encoded versions)**

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

FOR j ← 0 TO KL-1

i ← j \* 32

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

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

THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0];

ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];

FI;

ELSE

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

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

ELSE \*zeroing-masking\* ; zeroing-masking

DEST[i+31:i] = 0

FI

FI;

ENDFOR

DEST[MAXVL-1:VL] ← 0

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

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]

DEST[159:128] ← SRC1[159:128] BITWISE XOR SRC2[159:128]

DEST[191:160] ← SRC1[191:160] BITWISE XOR SRC2[191:160]

DEST[223:192] ← SRC1[223:192] BITWISE XOR SRC2[223:192]

DEST[255:224] ← SRC1[255:224] BITWISE XOR SRC2[255:224].

DEST[MAXVL-1:256] ← 0

**VXORPS (VEX.128 encoded version)**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]

DEST[MAXVL-1:128] ← 0

**XORPS (128-bit Legacy SSE version)**

DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]

DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]

DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]

DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]

DEST[MAXVL-1:128] (Unmodified)

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

VXORPS \_\_m512 \_\_mm512\_xor\_ps (\_\_m512 a, \_\_m512 b);

VXORPS \_\_m512 \_\_mm512\_mask\_xor\_ps (\_\_m512 a, \_\_mmask16 m, \_\_m512 b);

VXORPS \_\_m512 \_\_mm512\_maskz\_xor\_ps (\_\_mmask16 m, \_\_m512 a);

VXORPS \_\_m256 \_\_mm256\_xor\_ps (\_\_m256 a, \_\_m256 b);

VXORPS \_\_m256 \_\_mm256\_mask\_xor\_ps (\_\_m256 a, \_\_mmask8 m, \_\_m256 b);

VXORPS \_\_m256 \_\_mm256\_maskz\_xor\_ps (\_\_mmask8 m, \_\_m256 a);

XORPS \_\_m128 \_\_mm\_xor\_ps (\_\_m128 a, \_\_m128 b);

VXORPS \_\_m128 \_\_mm\_mask\_xor\_ps (\_\_m128 a, \_\_mmask8 m, \_\_m128 b);

VXORPS \_\_m128 \_\_mm\_maskz\_xor\_ps (\_\_mmask8 m, \_\_m128 a);

### **SIMD Floating-Point Exceptions**

None

### **Other Exceptions**

Non-EVEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded instructions, see Exceptions Type E4.

## XRSTOR—Restore Processor Extended States

Opcode / Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF AE /5 XRSTOR <i>mem</i>	M	V/V	XSAVE	Restore state components specified by EDX:EAX from <i>mem</i> .
NP REX.W + OF AE /5 XRSTOR64 <i>mem</i>	M	V/N.E.	XSAVE	Restore state components specified by EDX:EAX from <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.8, “Operation of XRSTOR,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP\_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If  $RFBM[i] = 0$ , XRSTOR does not update state component  $i$ .<sup>1</sup>
- If  $RFBM[i] = 1$  and bit  $i$  is clear in the XSTATE\_BV field in the XSAVE header, XRSTOR initializes state component  $i$ .
- If  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ , XRSTOR loads state component  $i$  from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR\_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component  $i$  for which  $RFBM[i] = 1$  and  $XSTATE\_BV[i] = 1$ ; it tracks as modified any state component  $i$  for which  $RFBM[i] = 0$ .

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR\_INFO.

1. There is an exception if  $RFBM[1] = 0$  and  $RFBM[2] = 1$ . In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

## Operation

```
RFBM ← XCRO AND EDX:EAX; /* bitwise logical AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
```

```
IF COMPMASK[63] = 0
  THEN
    /* Standard form of XRSTOR */
    TO_BE_RESTORED ← RFBM AND RSTORMASK;
    TO_BE_INITIALIZED ← RFBM AND NOT RSTORMASK;

    IF TO_BE_RESTORED[0] = 1
      THEN
        load x87 state from legacy region of XSAVE area;
        XINUSE[0] ← 1;
      ELSIF TO_BE_INITIALIZED[0] = 1
        THEN
          initialize x87 state;
          XINUSE[0] ← 0;
    FI;

    IF RFBM[1] = 1 OR RFBM[2] = 1
      THEN load MXCSR from legacy region of XSAVE area;
    FI;

    IF TO_BE_RESTORED[1] = 1
      THEN
        load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR
        XINUSE[1] ← 1;
      ELSIF TO_BE_INITIALIZED[1] = 1
        THEN
          set all XMM registers to 0; // this step does not initialize MXCSR
          XINUSE[1] ← 0;
    FI;

    FOR i ← 2 TO 62
      IF TO_BE_RESTORED[i] = 1
        THEN
          load XSAVE state component i at offset n from base of XSAVE area;
          // n enumerated by CPUID(EAX=0DH,ECX=i):EBX
          XINUSE[i] ← 1;
        ELSIF TO_BE_INITIALIZED[i] = 1
          THEN
            initialize XSAVE state component i;
            XINUSE[i] ← 0;
      FI;
    ENDFOR;

  ELSE
    /* Compacted form of XRSTOR */
    IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
      THEN /* compacted form not supported */
        #GP(0);
```

```

FI;

FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT ← RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        load x87 state from legacy region of XSAVE area;
        XINUSE[0] ← 1;
    ELSIF TO_BE_INITIALIZED[0] = 1
        THEN
            initialize x87 state;
            XINUSE[0] ← 0;
FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        XINUSE[1] ← 1;
    ELSIF TO_BE_INITIALIZED[1] = 1
        THEN
            set all XMM registers to 0;
            MXCSR ← 1F80H;
            XINUSE[1] ← 0;
FI;

NEXT_FEATURE_OFFSET = 576;           // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    XINUSE[i] ← 1;
                FI;
                NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
            FI;
            IF TO_BE_INITIALIZED[i] = 1
                THEN
                    initialize XSAVE state component i;
                    XINUSE[i] ← 0;
                FI;
        ENDFOR;
FI;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;

```

LAXA ← linear address of XSAVE area;  
 XRSTOR\_INFO ← (CPL,VMXNR,LAXA,COMPMASK);

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

XRSTOR: void\_xrstor( void \*, unsigned \_\_int64);  
 XRSTOR: void\_xrstor64( void \*, unsigned \_\_int64);

## Protected Mode Exceptions

#GP(0)	<p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.</p> <p>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.</p> <p>If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## Real-Address Mode Exceptions

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.</p> <p>If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.</p> <p>If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.</p> <p>If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p>
-----	---

If the compacted form is executed and a bit in the XCOMP\_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE\_BV field is 1.

If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

If attempting to write any reserved bits of the MXCSR register with 1.

#NM

If CR0.TS[bit 3] = 1.

#UD

If CPUID.01H:ECX.XSAVE[bit 26] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)

If a memory address is in a non-canonical form.

If a memory operand is not aligned on a 64-byte boundary, regardless of segment.

If bit 63 of the XCOMP\_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.

If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE\_BV field of the XSAVE header is 1.

If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.

If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP\_BV field of the XSAVE header is 1.

If the compacted form is executed and a bit in the XCOMP\_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE\_BV field is 1.

If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

If attempting to write any reserved bits of the MXCSR register with 1.

#SS(0)

If a memory address referencing the SS segment is in a non-canonical form.

#PF(fault-code)

If a page fault occurs.

#NM

If CR0.TS[bit 3] = 1.

#UD

If CPUID.01H:ECX.XSAVE[bit 26] = 0.

If CR4.OSXSAVE[bit 18] = 0.

If the LOCK prefix is used.

#AC

If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).



## XRSTORS—Restore Processor Extended States Supervisor

Opcode / Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF C7 /3 XRSTORS <i>mem</i>	M	V/V	XSS	Restore state components specified by EDX:EAX from <i>mem</i> .
NP REX.W + OF C7 /3 XRSTORS64 <i>mem</i>	M	V/N.E.	XSS	Restore state components specified by EDX:EAX from <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

### Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32\_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.12, “Operation of XRSTORS,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, “Extended Region of an XSAVE Area”).
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32\_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE\_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE\_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR\_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE\_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR\_INFO.

**Operation**

```

RFBM ← (XCRO OR IA32_XSS) AND EDX:EAX;          /* bitwise logical OR and AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;

FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT ← RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        load x87 state from legacy region of XSAVE area;
        XINUSE[0] ← 1;
    ELSIF TO_BE_INITIALIZED[0] = 1
        THEN
            initialize x87 state;
            XINUSE[0] ← 0;
FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        XINUSE[1] ← 1;
    ELSIF TO_BE_INITIALIZED[1] = 1
        THEN
            set all XMM registers to 0;
            MXCSR ← 1F80H;
            XINUSE[1] ← 0;
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    XINUSE[i] ← 1;
                FI;
                NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
            FI;
            IF TO_BE_INITIALIZED[i] = 1
                THEN
                    initialize XSAVE state component i;
                    XINUSE[i] ← 0;
                FI;
        ENDFOR;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation

```

```

THEN VMXNR ← 1;
ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← (CPL,VMXNR,LAXA,COMPMASK);

```

### Flags Affected

None.

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

```

XRSTORS:   void _xrstors( void *, unsigned __int64);
XRSTORS64: void _xrstors64( void *, unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	<p>If CPL &gt; 0.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

**Real-Address Mode Exceptions**

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	<p>If CPL &gt; 0.</p> <p>If a memory address is in a non-canonical form.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If bit 63 of the XCOMP_BV field of the XSAVE header is 0.</p> <p>If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.</p> <p>If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.</p> <p>If bytes 63:16 of the XSAVE header are not all zero.</p> <p>If attempting to write any reserved bits of the MXCSR register with 1.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## XSAVE—Save Processor Extended States

Opcode / Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF AE /4 XSAVE <i>mem</i>	M	V/V	XSAVE	Save state components specified by EDX:EAX to <i>mem</i> .
NP REX.W + OF AE /4 XSAVE64 <i>mem</i>	M	V/N.E.	XSAVE	Save state components specified by EDX:EAX to <i>mem</i> .

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCRO.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.7, “Operation of XSAVE,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component *i* if and only if  $RFBM[i] = 1$ .<sup>1</sup>
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVE reads the XSTATE\_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If  $RFBM[i] = 1$ , XSAVE writes XSTATE\_BV[*i*] with the value of XINUSE[*i*]. (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) If  $RFBM[i] = 0$ , XSAVE writes XSTATE\_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE\_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

$RFBM \leftarrow XCRO \text{ AND } EDX:EAX$ ; /\* bitwise logical AND \*/

$OLD\_BV \leftarrow XSTATE\_BV$  field from XSAVE header;

IF  $RFBM[0] = 1$

    THEN store x87 state into legacy region of XSAVE area;

FI;

IF  $RFBM[1] = 1$

1. An exception is made for MXCSR and MXCSR\_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either  $RFBM[1]$  or  $RFBM[2]$  is 1.

```

THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 OR RFBM[2] = 1
  THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
  IF RFBM[i] = 1
    THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
  FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);

```

### Flags Affected

None.

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

```

XSAVE:    void _xsave( void *, unsigned __int64);
XSAVE:    void _xsave64( void *, unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

### Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

### Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

### Compatibility Mode Exceptions

Same exceptions as in protected mode.

### 64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## XSAVEC—Save Processor Extended States with Compaction

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF C7 /4 XSAVEC <i>mem</i>	M	V/V	XSAVEC	Save state components specified by EDX:EAX to <i>mem</i> with compaction.
NP REX.W + OF C7 /4 XSAVEC64 <i>mem</i>	M	V/N.E.	XSAVEC	Save state components specified by EDX:EAX to <i>mem</i> with compaction.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCRO.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.10, “Operation of XSAVEC,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component *i* if and only if  $RFBM[i] = 1$  and  $XINUSE[i] = 1$ .<sup>1</sup> (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”)
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE\_BV field of the XSAVE header.<sup>2,3</sup> (See Section 13.4.2, “XSAVE Header.”) XSAVEC sets bit 63 of the XCOMP\_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE\_BV and XCOMP\_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

### Operation

$RFBM \leftarrow XCRO \text{ AND } EDX:EAX;$  /\* bitwise logical AND \*/

$TO\_BE\_SAVED \leftarrow RFBM \text{ AND } XINUSE;$  /\* bitwise logical AND \*/

If  $MXCSR \neq 1F80H$  AND  $RFBM[1]$

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE\_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE\_BV[1] to 1 as long as RFBM[1] = 1.



```

    TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;           // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
        FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← RFBM OR 80000000_00000000H;

```

### Flags Affected

None.

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

```

XSAVEC:    void _xsavvec( void *, unsigned __int64);
XSAVEC64:  void _xsavvec64( void *, unsigned __int64);

```

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

**Real-Address Mode Exceptions**

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

## XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP OF AE /6 XSAVEOPT <i>mem</i>	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to <i>mem</i> , optimizing if possible.
NP REX.W + OF AE /6 XSAVEOPT64 <i>mem</i>	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to <i>mem</i> , optimizing if possible.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCRO.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.9, “Operation of XSAVEOPT,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if  $RFBM[i] = 1$  and  $XINUSE[i] = 1$ .<sup>1</sup> (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR\_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVEOPT reads the XSTATE\_BV field of the XSAVE header (see Section 13.4.2, “XSAVE Header”) and writes a modified value back to memory as follows. If  $RFBM[i] = 1$ , XSAVEOPT writes  $XSTATE\_BV[i]$  with the value of  $XINUSE[i]$ . If  $RFBM[i] = 0$ , XSAVEOPT writes  $XSTATE\_BV[i]$  with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE\_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR\_INFO.

### Operation

$RFBM \leftarrow XCRO \text{ AND } EDX:EAX; /* \text{ bitwise logical AND } */$

1. There is an exception made for MXCSR and MXCSR\_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if  $RFBM[1] = 1$  or  $RFBM[2] = 1$ , regardless of the value of XINUSE.

OLD\_BV ← XSTATE\_BV field from XSAVE header;  
 TO\_BE\_SAVED ← RFBM AND XINUSE;

IF in VMX non-root operation

    THEN VMXNR ← 1;  
     ELSE VMXNR ← 0;

FI;

LAXA ← linear address of XSAVE area;

IF XRSTOR\_INFO = ⟨CPL,VMXNR,LAXA,00000000\_00000000H⟩  
     THEN TO\_BE\_SAVED ← TO\_BE\_SAVED AND XMODIFIED;

FI;

IF TO\_BE\_SAVED[0] = 1

    THEN store x87 state into legacy region of XSAVE area;

FI;

IF TO\_BE\_SAVED[1]

    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR\_MASK

FI;

IF RFBM[1] = 1 or RFBM[2] = 1

    THEN store MXCSR and MXCSR\_MASK into legacy region of XSAVE area;

FI;

FOR i ← 2 TO 62

    IF TO\_BE\_SAVED[i] = 1

        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);

    FI;

ENDFOR;

XSTATE\_BV field in XSAVE header ← (OLD\_BV AND NOT RFBM) OR (XINUSE AND RFBM);

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

XSAVEOPT: void \_xsaveopt( void \*, unsigned \_\_int64);

XSAVEOPT: void \_xsaveopt64( void \*, unsigned \_\_int64);

## Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

**Real-Address Mode Exceptions**

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

## XSAVES—Save Processor Extended States Supervisor

Opcode / Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C7 /5 XSAVES <i>mem</i>	M	V/V	XSS	Save state components specified by EDX:EAX to <i>mem</i> with compaction, optimizing if possible.
NP REX.W + 0F C7 /5 XSAVES64 <i>mem</i>	M	V/N.E.	XSS	Save state components specified by EDX:EAX to <i>mem</i> with compaction, optimizing if possible.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

### Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32\_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, “XSAVE Area,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*.

Section 13.11, “Operation of XSAVES,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32\_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.<sup>1</sup> (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, “Processor Tracking of XSAVE-Managed State.”) Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR\_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, “Legacy Region of an XSAVE Area”).
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE\_BV field of the XSAVE header.<sup>2</sup> (See Section 13.4.2, “XSAVE Header.”) XSAVES sets bit 63 of the XCOMP\_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE\_BV and XCOMP\_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, “Extended Region of an XSAVE Area”).

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, “Processor Tracking of XSAVE-Managed State,” of *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR\_INFO.

- 
1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
  2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE\_BV[1] to 1 as long as RFBM[1] = 1.

## Operation

```

RFBM ← (XCRO OR IA32_XSS) AND EDX:EAX;          /* bitwise logical OR and AND */
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← RFBM OR 80000000_00000000H;
TO_BE_SAVED ← RFBM AND XINUSE;
IF XRSTOR_INFO = ⟨CPL,VMXNR,LAXA,COMPMASK⟩
    THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;
IF MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN
                    save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    IF i = 8          // state component 8 is for PT state
                        THEN IA32_RTIT_CTL.TraceEn[bit 0] ← 0;
                    FI;
                FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
        FI;
    ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← COMPMASK;

```

## Flags Affected

None.

## Intel C/C++ Compiler Intrinsic Equivalent

```

XSAVES:    void _xsaves( void *, unsigned __int64);
XSAVES64:  void _xsaves64( void *, unsigned __int64);

```

**Protected Mode Exceptions**

#GP(0)	<p>If CPL &gt; 0.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

**Real-Address Mode Exceptions**

#GP	<p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p> <p>If any part of the operand lies outside the effective address space from 0 to FFFFH.</p>
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>

**Virtual-8086 Mode Exceptions**

Same exceptions as in protected mode.

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.



## 64-Bit Mode Exceptions

#GP(0)	<p>If CPL &gt; 0.</p> <p>If the memory address is in a non-canonical form.</p> <p>If a memory operand is not aligned on a 64-byte boundary, regardless of segment.</p>
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>
#AC	<p>If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 64-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).</p>

## XSETBV—Set Extended Control Register

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP 0F 01 D1	XSETBV	Z0	Valid	Valid	Write the value in EDX:EAX to the XCR specified by ECX.

### Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
Z0	NA	NA	NA	NA

### Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, "Enabling the XSAVE Feature Set and XSAVE-Enabled Features," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

### Operation

XCR[ECX] ← EDX:EAX;

### Flags Affected

None.

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

XSETBV: `void _xsetbv(unsigned int, unsigned __int64);`

### Protected Mode Exceptions

#GP(0)	<ul style="list-style-type: none"> <li>If the current privilege level is not 0.</li> <li>If an invalid XCR is specified in ECX.</li> <li>If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.</li> <li>If an attempt is made to clear bit 0 of XCR0.</li> <li>If an attempt is made to set XCR0[2:1] to 10b.</li> </ul>
#UD	<ul style="list-style-type: none"> <li>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</li> <li>If CR4.OSXSAVE[bit 18] = 0.</li> <li>If the LOCK prefix is used.</li> </ul>

**Real-Address Mode Exceptions**

#GP	<p>If an invalid XCR is specified in ECX.</p> <p>If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.</p> <p>If an attempt is made to clear bit 0 of XCR0.</p> <p>If an attempt is made to set XCR0[2:1] to 10b.</p>
#UD	<p>If CPUID.01H:ECX.XSAVE[bit 26] = 0.</p> <p>If CR4.OSXSAVE[bit 18] = 0.</p> <p>If the LOCK prefix is used.</p>

**Virtual-8086 Mode Exceptions**

#GP(0)	The XSETBV instruction is not recognized in virtual-8086 mode.
--------	--

**Compatibility Mode Exceptions**

Same exceptions as in protected mode.

**64-Bit Mode Exceptions**

Same exceptions as in protected mode.

## XTEST – Test If In Transactional Execution

Opcode/Instruction	Op/En	64/32bit Mode Support	CPUID Feature Flag	Description
NP OF 01 D6 XTEST	A	V/V	HLE or RTM	Test if executing in a transactional region

### Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

### Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

### Operation

#### XTEST

```
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
  THEN
    ZF ← 0
  ELSE
    ZF ← 1
```

FI;

### Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

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

```
XTEST:   int_xtest(void);
```

### SIMD Floating-Point Exceptions

None

### Other Exceptions

#UD CPUID.(EAX=7, ECX=0):EBX.HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0.  
If LOCK prefix is used.